RSFQ Digital Circuit Design Automation and Optimisation

by

Louis C. Müller

Dissertation presented for the degree of Doctor of Philosophy in Electronic Engineering in the Faculty of Engineering at Stellenbosch University

Promoter:

Prof. Coenrad J. Fourie

Department of Electrical and Electronic Engineering
University of Stellenbosch
Stellenbosch, South Africa

March 2015

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: .....

Abstract

In order to facilitate the creation of complex and robust RSFQ digital logic circuits an extensive library of electronic design automation (EDA) tools is a necessity. It is the aim of this work to introduce various methods to improve the current state of EDA in RSFQ circuit design.

Firstly, Monte Carlo methods such as Latin Hypercube sampling and Sobol sequences are applied for their variance reduction abilities in approximating circuit yield. In addition, artificial neural networks are also investigated for their applicability in modeling the parameter-yield space.

Secondly, a novel technique for circuit functional testing using automated state machine extraction is presented, which greatly simplifies the logical verification of a circuit. This method is also used, along with critical timing extraction, to automatically generate Hardware Description Language(HDL) models which can be used for high level circuit design.

Lastly, the Greedy Local search, Simulated Annealing and Genetic Algorithm meta-heuristics were statistically compared in a novel manner using a yield model provided by artificial neural networks. This is done to ascertain their performance in optimising RSFQ circuits in relation to yield.

The variance reduction techniques of Latin Hypercube Sampling and Sobol sequences were shown to be beneficial for the use with RSFQ circuits. For optimisation purposes the use of Simulated Annealing and Genetic Algorithms were shown to improve circuit optimisation for possible multi-modal search spaces. An HDL model is also successfully generated from a complex RSFQ circuit for use in high level circuit design which includes critical timing and propagation latency.

All the techniques presented in this study form part of a software library that can be further refined and extended in future work.

Acknowledgements

My gratitude must always go foremost to the Almighty for granting me the ability and perseverance to complete this project while being employed as an engineer in the industry.

I would also like to express my sincerest gratitude to the following people:

To my unbelievable wife who continued to encourage me through late nights, early mornings and looming work deadlines, all the while raising a very busy Little Man. My gratitude and love for you cannot be expressed in prose or the most complex of quantum mechanical equations.

Prof. C. J. Fourie, my supervisor at the University of Stellenbosch, for his patience and guidance throughout this whole project. Thank you, especially, for understanding when work sometimes interfered with studies and for granting me the freedom to investigate various weird and wonderful research avenues (which might not all have been successful).

The various open source projects that allowed the investigation and software creation for this project: CodeLite, Boost, FANN, Matplotlib and the work of S. Joe and F. Y. Kuo on Sobol sequences.

Thank you also to Dr. N. J. Els at the University of Stellenbosch for creating the LaTeXtemplates that have been used to compile this thesis.

To Tristan Goss, my employer, thank you for allowing me the time to complete this project.

For those members of my family who have already passed on. Your influence on my life, though far too short, can never be understated. Your example will remain with me forever.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

RSFQ	Rapid Single Flux Quantum
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
ASIC	Application Specific Integrated Circuit
HDL	Hardware Description Language
EDA	Electronic Design Automation
HTS	High Temperature Superconductor
LTS	Low Temperature Superconductor
FF	Flip-Flop
DC	Direct Current
SFQ	Single Flux Quantum
JTL	Josephson Transmission Line
SQUID	Superconduting Quantum Interference Device
CMC	Crude Monte Carlo
ANN	Artificial Neural Network
ADALINE	Adaptive Linear Elements
RTL	Register Transfer Level
GLS	Greedy Local Search
SA	Simulated Annealing
GA	Genetic Algorithm

Chapter 1

Introduction

The semiconductor industry has been advancing at a staggering rate since integrated circuits became a commercial reality, mostly fuelled by an ever increasing human need for processing power. In order to serve this need, world leaders in integrated technology such as Intel and AMD have been pushing the limits in terms of circuit performance. The competition for the highest clock frequency might have been set aside due to the constraints of physics, but the process node size continues to shrink, allowing more transistors to consume less power on the same die size. Understandably, this progress has not been simple, with current leaders, Intel, spending billions of dollars on finalising their latest 14 nm FinFET technology while already working on a 10 nm node size. One might wonder how long the current node reduction trend can continue before a replacement for the silicon-based devices becomes a necessity.

Possible replacements are the various digital logic families that use superconductive state of materials. One of the most promising of these families is the Rapid Single Flux Quantum (RSFQ) [1] devices which exhibit the novel behaviour of using the smallest unit of magnetic flux as its unit of information storage and propagation. Complex circuits employing this logic family and operating at frequencies far above the capabilities of semiconductor devices have already been tested [2, 3]. There are, however, immense challenges facing the RSFQ community before this technology can become part of the mainstream.

The key obstacle remains the superconductive condition itself. Although superconductivity has been shown to occur in materials at above 100 Kelvin [4], the materials used for these achievements are not easily manufactured in integrated circuits. Low temperature superconductors, such as Niobium, with critical temperatures of around 4 Kelvin, are therefore still predominantly employed in the superconducting digital logic industry. Moreover, expensive cryogenic equipment is necessary to retain the circuits in their operating region. Research in higher temperature superconductors and cryogenic equipment is still ongoing.

Another obstacle facing the RSFQ community is high operating frequency itself. At the tens of gigahertz where these circuits operate, clock skew evolves

into a substantial problem. In effect, the circuit operates at frequencies that are comparable to the propagation delay of the information carriers themselves, which makes timing closure very difficult. However, some novel approaches to timing and asynchronous methodologies [5] have been proposed which might be more suited to this technology.

The challenges faced when designing timing critical, complex circuits are not unique to RSFQ though, it is a problem that has already been researched for decades in the semiconductor industry. In order to alleviate such design challenges, various pieces of software, called electronic design automation software, were developed. These software packages permit the automation of various steps in the design process in order to allow designers to create more complex circuitry without the need to manually follow through with the complete implementation. Such software packages are currently an essential part of the circuit design process, be it in the arena of the Field Programmable Gate Array(FPGA) or the Application Specific Integrated Circuit (ASIC).

The aim of the presented work is to investigate the current state of RSFQ electronic design automation in order to propose various fundamental improvements to the design methodology. A secondary deliverable is also provided in the form of a software library that contains the methodological improvements presented in this work.

1.1 Current state of RSFQ Electronic Design Automation

Although the design of superconductor electronic circuits has matured significantly in the past few decades, most of the steps still necessitate a substantial amount of human input. This is in stark contrast to the current state of circuit design in the semi-conductor industry where a very high level of abstraction is available to designers so that they can focus on the specific function being implemented rather than the details of the implementation itself. An obvious example is the high level synthesis provided by Xilinx [6] where applications can be written in C, C++ or SystemC and the provided software facilitates a direct implementation to the Hardware Description Language(HDL) level, and in this case, the targeted FPGA. The MathWorks [7] arguably takes the process a step further by supplying a block diagram-based environment (Simulink) from which an HDL implementation can be generated and after which a vendor specific synthesis process can commence.

It was shown in [8] that the state of Electronic Design Automation(EDA) software used for superconducting circuit development tends to be substantially fragmented, with many institutions using in-house solutions that have not been updated for a relatively long period of time. Therefore, opportunities abound to improve the current state of EDA software in the RSFQ industry.

In order to keep the scope of the presented work tractable, two areas that are in need of improvement were selected for further investigation. The first area was that of circuit yield optimisation. Yield designates the percentage of manufactured circuits that are deemed to be functional. The optimisation of yield is therefore crucial for the creation of robust circuits that can form part of a larger design. Yield is generally calculated by using a Monte Carlo method that relies on many simulation runs. The yield calculations and optimisation therefore tend to be time-consuming.

The current state of EDA software in the superconductor electronics industry employed in yield optimisation can be summarised as follows:

- In order to calculate the circuit yield, pass or fail criteria need to be created. This is generally achieved manually (usually using a SPICE test bench).
- To ascertain whether a circuit is functional or not, results from the previously mentioned test bench are analysed in terms of junction switches and output pulses. This is achieved either manually or by using a programming script.
- A Crude Monte Carlo approximation is generally used for yield calculations which necessitates many simulation runs in order to obtain a usable confidence interval.
- Numerous yield optimisation algorithms are actively used, mostly with a stochastic foundation. It is therefore very challenging to compare different algorithms and different variants of the same algorithm.

The second area under investigation was the creation of Hardware Description Language (HDL) models for high level circuit design. A Hardware Description Language is a programmatic input method used to describe circuit functionality and circuit interconnections. The usage of HDL has become a part of the mainstream in the semiconductor industry due to the ease with which complex circuitry can be described. The circuit HDL description along with a robust cell library lends itself to a fast and repeatable design cycle. As is indicated in [8], many methods of HDL modeling have already been proposed. However, these models tend to be constructed using various different methodologies which can be time consuming. Furthermore, no standard exist for the creation of these models making the inter-institution exchange of cell libraries difficult.

The current state of EDA software in the superconductor electronics industry employed in HDL model creation is as follows:

- State machine representations are created manually or via script by monitoring circuit junction switches and output pulse generation.

- Timing characterisation is achieved either manually or via script by monitoring circuit junction switches and output pulse generation.
- Model construction methodologies tend to be institution specific.

The aim of this work is to propose various methods to improve the state of EDA software in the superconducting electronics industry by focusing on automating various steps in the design cycle and proposing algorithmic improvements to reduce the design time.

The aims are thus:

- To automate the creation of pass or fail test benches and interpret the results.
- To propose various methods for faster yield analysis.
- To propose a method for the comparison of different optimisation algorithms and the comparison of different variants of the same algorithms.
- To propose a method for the automated creation of an HDL model with its accompanying timing characterisation in a repeatable, standardised manner.

1.2 Document Layout

In Chapter 2 of this document a high level explanation of the phenomenon of superconductivity is presented. This is followed by a discussion on the functional behaviour of RSFQ circuits.

Chapter 3 proposes various Monte Carlo methods for approximating the process yield of an RSFQ circuit. Here, Latin Hypercube sampling and Sobol sequences are compared with Crude Monte Carlo in terms of their confidence interval.

Chapter 4 introduces artificial neural networks specifically for approximating circuit yield. The results of this section are discussed in Chapter 6, with its applicability to circuit optimisation.

Chapter 5 presents an automated state machine extraction algorithm which can be applied to RSFQ circuits for failure testing and HDL model generation. The timing characterisation of the circuits are also investigated for use with the HDL models.

In Chapter 6 various meta-heuristic optimisation algorithms are compared by using an artificial neural network for yield approximation. The artificial neural network is employed to facilitate vast numbers of optimisation runs in order to investigate the statistical performance of the meta-heuristics.

The conclusion and possible future work are presented in Chapter 7. An appendix is also attached which contains a broad overview of the software

library of the algorithms presented in this document and the results of the artificial neural network investigation.

Chapter 2

Background Information

2.1 History

Superconductivity as a physical phenomena was first discovered in 1911 by a Dutch physicist named Heike Kammerlingh Onnes [4], a pioneer in cryogenics. His aim was to observe the properties of metals in liquid helium. The understanding at the time was that the resistance should decrease by some continues function as the temperature decreases. When testing mercury however, the physicist identified a very sharp drop in temperature, down to zero ohm at around 4 Kelvin, in effect a discontinuity that was not observed before. Soon many other metals exhibiting the same behaviour was found. The temperature where this sudden transition to zero ohm occurred was dubbed the critical temperature T_c. For a long time the metal with the highest T_c was Niobium, with a value of 9 Kelvin. However, in 1987 a new type of superconductor material was identified with T_c values far higher than previously recorded. Materials, such as $YBa_2Cu_3O_7$ with T_c values of around 92 Kelvin became known as High Temperature Superconductors (HTS) while the previously found materials were known as Low Temperature Superconductors(LTS).

As discussed in this document, it is evident that the usage of LTS materials is still very prevalent in the manufacturing of digital logic circuits. This is due to the ease of manufacturing with materials such as Niobium. In contrast, the manufacturing of HTS circuitry tends to pose various challenges which fall outside the scope of this work. It is therefore still necessary to employ relatively expensive cryo-coolers in order to test the LTS circuits.

2.2 Properties of Superconductors

The currently accepted theory behind superconductors, called the BCS theory (named after the J. Bardeen, L. Cooper and J. Shrieffer), makes heavy use of quantum mechanics to describe how and why superconductors function as they do. In this work, a more intuitive method is used to describe their behaviour by

using some of the properties of the BCS theory and following the explanations in [4]. Interested readers are directed to [9] for a more thorough investigation of the theory behind superconductivity.

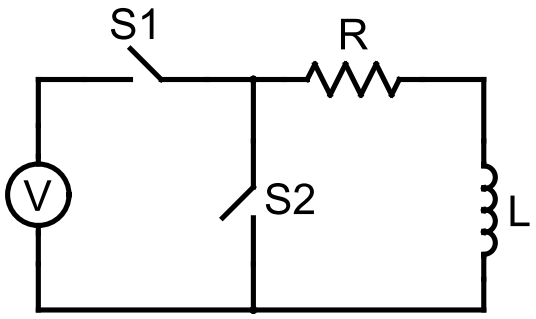


Figure 2.1: Superconductor as an ideal inductor.

A useful analogy to use when dealing with superconductors is to model the superconductor as an ideal inductor. An example is illustrated in Figure 2.1. Assuming that $S2$ is open and $S1$ is closed for a short period, energy will be stored in the circuit inductor as a magnetic field as long as these switch configurations are held constant. If, for example, the switch positions are reversed ($S1$ open and $S2$ closed), the current in the circuit will continue to flow and the stored energy will eventually be completely dissipated in the resistor. In superconductors, though, the resistance value is zero, which means that due to the stored magnetic field the current will continue to flow forever. This has been shown to be the case in practise, where no substantial changes were found when using superconductors in experiments where the current value was tested over a period of years. It is noteworthy, though, that superconductors have a current limit called the critical current, I_c. If a current level above this value is applied through the superconductor, the latter will enter its resistive state, and hence behave like a normal conductor.

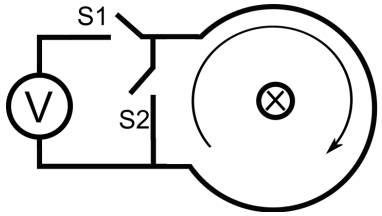


Figure 2.2: Superconductor as a flux insulator.

Another way of interpreting the superconductive state is to rather investigate the change in magnetic flux. In Figure 2.2 where the loop is of a material in a supeconductive state, the voltage source will continually add magnetic flux to the circuit due to $V = d\Phi/dt$ which is stored in the loop. The only

way for the flux to escape the loop is through a resistance which is zero due to the superconductive state. There is therefore no other means for the flux to leave the loop. If, for example, the voltage source is shorted and a magnet is brought close to the loop, the change in magnetic field due to the magnet will cause an opposing current to form in an attempt to counteract the new magnetic component (as is the principal behind electric generators). In a normal metal, as soon as the magnet stops moving (and the change in magnetic field returns to zero), this opposing current will decay with a time constant of L/R. In superconducting circuits, however, the resistance value is zero which indicates that the opposing current will continue to flow even if the applied magnetic field is static. The shielding current will therefore perpetually cause a counteractive magnetic field to nullify the flux component of the magnet. A superconductor can therefore also be regarded as a "flux insulator".

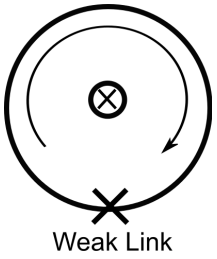


Figure 2.3: Weak link in a superconductor.

There are, however, ways in which flux can enter and leave a superconducting loop under certain conditions. For instance, assuming that a superconducting loop, as shown in Figure 2.3, contains a weak link. This weak link could, for example, be a narrowing of the superconducting wire, or a thin insulator connecting the superconductive wires. Under normal circumstances, the loop will hold its magnetic flux indefinitely, as explained above. If, however, the current flowing in the loop is larger than the critical current of the weak link, flux will be able to escape from the loop due to the weak link entering its resistive state. What is interesting to note is that instead of a continuous loss of flux, the flux will rather decrease by one fluxon at a time. A fluxon being the smallest quantised measure of magnetic flux with relation $\Phi = h/2e$ where h is the Plank constant and e the electron charge. Each time a flux leaves the loop, a small voltage pulse is created due to $V = d\Phi/dt$.

Another important concept in superconducting circuits is the quantisation of flux storage. Through the BCS theory, it has been established that the charge carrying particles in a superconductor are Cooper pairs rather than normal electrons. These are pairs of electrons coupled together as a result of electron-phonon interactions [9]. The theory states that not only should all these Cooper pairs be in the same energy state, but they should also be governed by the same quantum mechanical wave equation with $\Psi = \Psi_0 exp(j\theta)$.

In this equation, Ψ_0^2 is the number density of the Cooper pairs and θ is the phase of the wave function. In effect, all the wave functions of each Cooper pair add up coherently to form a macroscopic particle. As described in [4], when a voltage pulse with an integral equal to that of a single fluxon is applied to a superconducting loop, such as seen in Figure 2.3, a phase change of 2π is introduced around the loop so that the start and end phases of the wave function align. In effect, a continuous phase relationship is present around the loop without any discontinuous phase jumps. Any discontinuities in this phase relationship is energetically unfavourable given that the same wave function is used for all the Cooper pairs in the loop. It is thus evident from this discussion that only modula 2π phase changes are allowed inside the superconducting loop which is associated with the storage or loss of a single magnetic flux quantum.

2.3 Josephson Junctions

From the previous discussion it is evident that in order to connect two superconducting wires, the phases of the wave functions must align. A deviation from this condition might occur if a weak link, as depicted in Figure 2.3, is formed between the wires. Consequently, the wave functions will be weakly coupled and allow a difference in the phase across the link [4]. Such "weak links" are called Josephson junctions, taking their name from B.D. Josephson who first theorised that electron pairs will be able to tunnel between closely spaced superconductors [9]. The super current flowing through the junction is related to the difference in the phase of the two wave functions as presented in equation 2.3.1

$$I = I_c sin\phi \tag{2.3.1}$$

where $\phi = \theta_2 - \theta_1$. The change in junction phase is related to the voltage across the junction as shown in equation 2.3.2.

$$\frac{\partial\phi}{\partial t} = \frac{2eV}{\hbar} = \frac{2\pi V}{\Phi_0} \tag{2.3.2}$$

From equation 2.3.2, it can be surmised that if a constant voltage is applied across the junction, an oscillating current will be created with frequency $f_J = 2eV/\hbar = V/\Phi_0$.

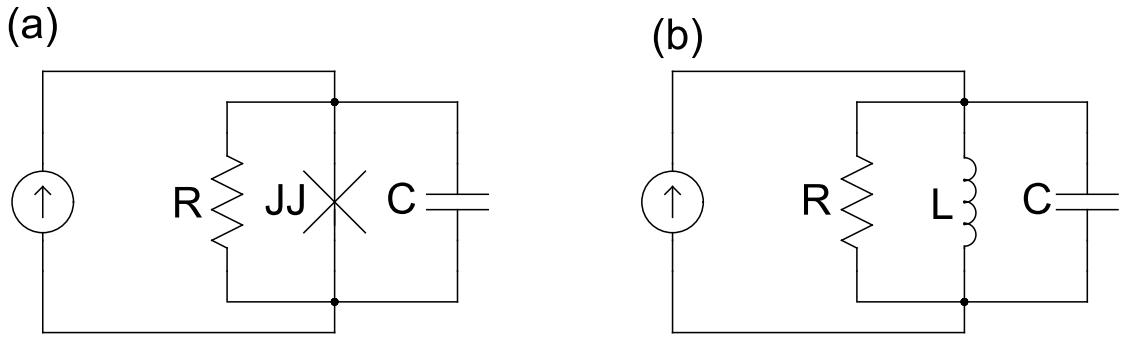


Figure 2.4: RCSJ model of Josephson junction with approximation

A popular method of modelling a Josephson junction is the use of the RCSJ model as depicted in Figure 2.4. In this model three current paths are illustrated. The resistive path is the normal current when the instantaneous current is larger than the junction critical current while the capacitor is formed from the capacitive structure ,superconductor-insulator-superconductor, of the junction itself. The total instantaneous current can therefore be written as [4]:

$$\begin{aligned} I = I_s + I_n + I_d &= I_c sin(\phi) + \frac{V}{R} + C\frac{dV}{dt} \\ &= I_c sin(\phi) + \frac{\hbar}{2eR}\frac{d\phi}{dt} + \frac{\hbar C}{2e}\frac{d^2\phi}{dt^2} \end{aligned} \tag{2.3.3}$$

where equation 2.3.2 is used to replace V. Since equation 2.3.3 is a non-linear second-order differential equation, a linearisation around ϕ is used to gather further insight into the model. This results in the following:

$$I = \frac{\Phi'}{L_{J0}} + \frac{V}{R} + C\frac{dV}{dt} \tag{2.3.4}$$

where $\phi' = \Phi_0\phi/2\pi$, $V = d\Phi'/dt$. $L_{J0} = \Phi_0/2\pi = \hbar/2eI_c$ is the limit of the junction inductance as $I_s << I_c$ as was derived in [4].

As is presented in [4], this describes a current-driven parallel RCL resonator with resonant frequency (sometimes called the plasma frequency in superconducting literature) of:

$$\omega_0 = 2\pi f_0 = \frac{1}{(L_{J0}C)^{1/2}} \tag{2.3.5}$$

and a Q value of

$$Q = \omega RC = \left(\frac{R^2C}{L_{J0}}\right)^{1/2} \tag{2.3.6}$$

Analogous to the resonator, the Q value also describes how "damped" the junction is, which plays a major role in the dynamic behaviour of the junction

as will be explained below. The Q value itself is rarely used in superconducting literature but rather re-derived as $\mathcal{B}_c$ as indicated in equation 2.3.7.

$$\mathcal{B}_c = Q^2 = \frac{2eR^2CI_c}{\hbar} \tag{2.3.7}$$

It is interesting to note that equation 2.3.3 is analogous to that of a pendulum system, as described in [4, 9]. In general a pendulum is governed by equation 2.3.8

$$T_{tot} = \mu\frac{d^2\phi}{dt^2} = T_a - ksin(\phi) - \nu\frac{d\phi}{dt} \tag{2.3.8}$$

where T_a is the applied torque, $\mu = ml^2$ is the moment of inertia, $d^2\phi/dt^2$ is the angular acceleration and $k = mgl$ is the restoring constant. For each of these components of the pendulum there is an analogous component in the junction model of equation 2.3.3. This is presented in Table 2.1 which has been reproduced from [4].

Figure 2.5 illustrates the pendulum analogy.

Junction Components	Pendulum Components
Phase difference ϕ	Angle from vertical ϕ
Voltage $V = (\hbar/2e)d\phi/dt$	Angular Velocity $d\phi/dt$
Capacitance C	Moment of inertia $\mu = ml^2$
Conductance $1/R$	Viscosity ν
Critical Current I_c	Restoring constant $k = mgl$
Applied Current I	Applied Torque T_a
Plasma Frequency $\omega_0 = 1/\sqrt{L_{J0}C}$	Oscillating Frequency $\omega_0 = \sqrt{k/\mu}$

Table 2.1: Analogy between Josephson junction parameters and pendulum parameters

Bearing in mind the value of Q (or $\mathcal{B}_C$), an intuitive understanding of the junction behaviour can now be gained. Assuming that the pendulum starts at rest with $\phi = 0$, then the applied torque will increase the angular velocity and commence the rotation of the pendulum around its anchor point. If less torque than the restoring constant is applied and removed, gravity and viscosity will ensure that the pendulum returns to its nominal position. This might include some oscillations around the origin depending on the Q value of the pendulum system. If $Q >> 1/2$, the pendulum will oscillate substantially before coming to rest (under-damped). If $Q << 1/2$, the pendulum will not oscillate at all (over-damped). If the applied torque pulse is larger than the restoring constant, the pendulum will swing around and rotate around its anchor point, oscillating again depending on the Q value.

The behaviour of the Josephson junction is very similar. If a current weaker than the critical current value is applied and removed through the junction, the

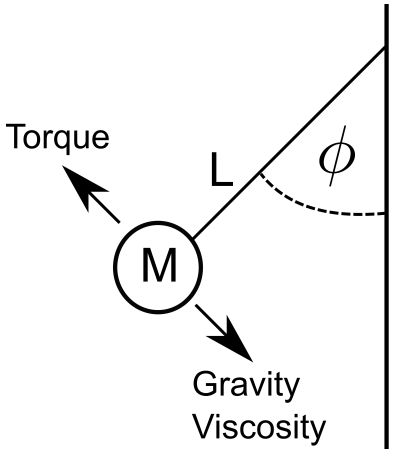


Figure 2.5: Illustration of Josephson junction pendulum analogy

junction phase will increase and decrease back to its original value, possibly oscillating depending on the value of $\mathcal{B}_c$. If, however, a current above the critical current value is applied, the junction will "switch" thus changing its phase around 2π exactly like the pendulum equivalent. As indicated previously, this 2π phase shift is equivalent to a single fluxon passing over the junction.

2.4 Rapid Single Flux Quantum Circuits

Some of the first implementations of logical circuits using the Josephson junction as active elements attempted to emulate the voltage state logic as used in the semiconductor industry. A very high profile project, headed by IBM [10] attempted to implement a Josephson computer using voltage state logic. The implementation employed under-damped junctions exhibiting hysteresis in their resistive state. For example, when a current above the critical current value is applied to the junction, the junction remains in its resistive state until the current is removed completely (not only reduced to a level below the critical current). The junction therefore forms a simple switch that can be set and reset through the bias current applied to the junction. A logical level of 1 was then associated with the junction being in its resistive state and a logical level of 0 was associated with the superconducting state. This project was eventually terminated due to implementation difficulties as well as due to the advancement of the semiconductor industry. Various other voltage state logic families were also tested and implemented with better results [11, 12].

In contrast to these voltage-based logical families, Lihkarev et al. [1] proposed a logical family that rather used the absence or presence of single flux quantum pulses as indication of a logical 1 or 0. Thus instead of propagating voltage levels through the system, fluxons were used to change circuit behaviour and propagate information. This is intuitively a very pleasing solution since, as explained earlier, the switching of a Josephson junction releases

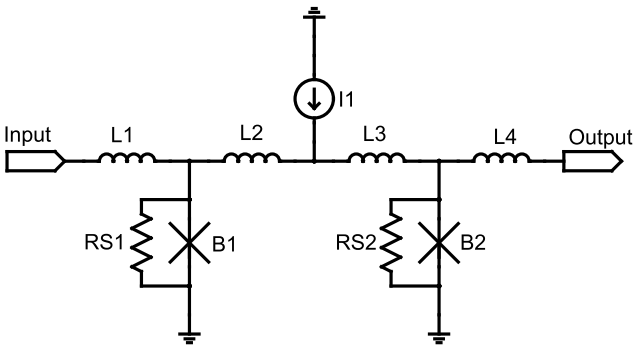


Figure 2.6: Circuit schematic of a Josephson transmission line.

SFQ pulses via its normal switch operation. Instead of using under-damped junctions with their hysteric behaviour, over-damped junctions are employed. To understand the reasoning begin this, one needs to consider the pendulum analogy as depicted in Figure 2.5. Assuming that a DC bias current is applied to the junction (a constant torque in the pendulum analogy), this will ensure that the pendulum is situated at a certain position of $\phi > 0$. If a SFQ pulse is then applied to the junction, the small associated current pulse could be sufficient to drive the current level over its critical threshold. In the pendulum analogy this would signify the application of a small torque that is sufficient to rotate the pendulum above the critical angle of $\phi > \pi$, causing the pendulum to undergo a revolution. As described earlier, for the junction this indicates that another SFQ pulse was generated and is free to propagate onwards through the system. Due to the over-damped nature of the system, the junction quickly returns to its biased position (caused by the DC bias current) without the need to remove the bias current, thus leaving the junction ready to propagate the next SFQ pulse. It is therefore evident that a junction can be used to propagate SFQ pulses throughout a circuit. This logical family is called the Rapid-Single-Flux-Quantum (RSFQ) family.

An example of such a circuit is illustrated in Figure 2.6.

In the circuit I_1 is used to bias junctions B_1 and B_2 to around 70% of their critical current value. The shunt resistors R_{S1} and R_{S2} are used to alter the $\mathcal{B}_c$ value shown in equation 2.3.7 to approximately 1, thereby forming a over-damped junction. If an SFQ pulse arrives at the input, sufficient current is applied to junction B_1 to temporarily switch it to its resistive state. This causes a fluxon (SFQ pulse) to be propagated into the circuit. The current direction created as a result of the propagated fluxon is then clock-wise in the loop B_1—L_2—L_3—B_2. This newly released SFQ pulse will have an additive effect on the current through B_2, causing it to switch temporarily to its resistive state as well. Subsequently, another SFQ pulse will be released through B_2 and propagate to the output. Due to this type of behaviour this circuit is generally called the Josephson transmission line (JTL) and forms the core of RSFQ circuit interconnections.

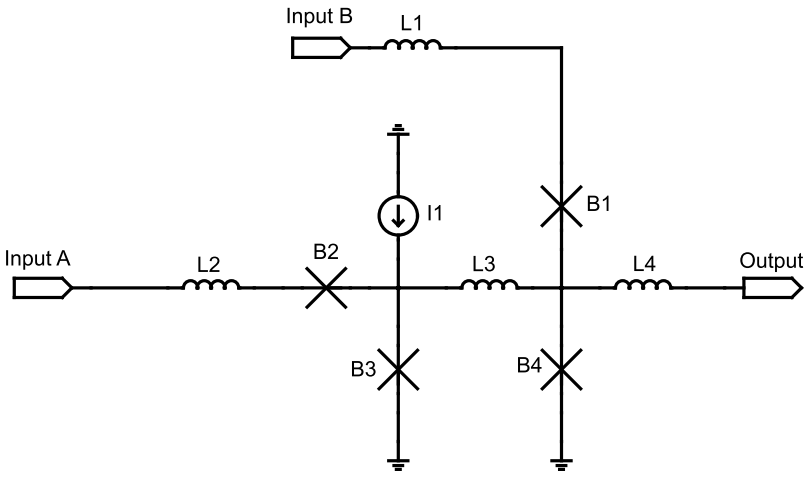


Figure 2.7: Circuit schematic of an RSFQ Destructive-Flip-Flop

Having explained the propagation of SFQ pulses, the next function which requires attention is the storage element. For this, another two junction circuit, called a DC SQUID, is used. These circuits are depicted in Figure 2.7 where the shunt resistances were omitted for the sake of clarity.

In Figure 2.7 junction B_3 is once again biased to the 70% mark in order to receive and propagate an SFQ pulse. Junction B_4 on the other hand, has a low bias given that the current is not equally split from I_1 due to the inductor L_3. If an SFQ pulse arrives at $InputA$, a switch occurs in B_3 due to its current biasing, as was described for the JTL implementation. Note that $\Phi = LI$, therefore the flux is equal to the inductance and current in the loop. Since the amount of flux entering the loop is known (that of a single fluxon), the amount of current generated in the loop is dependent on the loop inductance. If the selected value of L_3 is large enough, then a clock-wise current flow will be generated in the loop B_3—L_3—B_4 that is not sufficient to switch B_4 to its resistive state. The fluxon is therefore trapped inside the loop. In order to release this fluxon, a pulse at $InputB$ is necessary. The current caused by the SFQ pulse at $InputB$ is additive to the clock-wise current in the loop due to the trapped fluxon. Together these two currents are then sufficient to switch B_4 to its resistive state and propagate a fluxon to the output. The stored fluxon value is lost through the operation, hence the circuit is named a destructive-flip-flop.

In Figure 2.7, series junctions are also introduced. Such junctions are used to protect the circuit under all logical conditions. For example, if no fluxon is stored, B_1 should be biased to around 70% of its critical current. When a pulse arrives at $InputB$ this should cause B_1 to switch, therefore "throwing out" the entered SFQ pulse from the circuit. If a fluxon is stored in the loop, some of the clock-wise current due to the stored fluxon is shunted in the direction from the loop to $InputB$. This has the subtractive effect of lowering the bias current on B_1. A pulse at $InputB$ now does not supply sufficient current for B_1 to

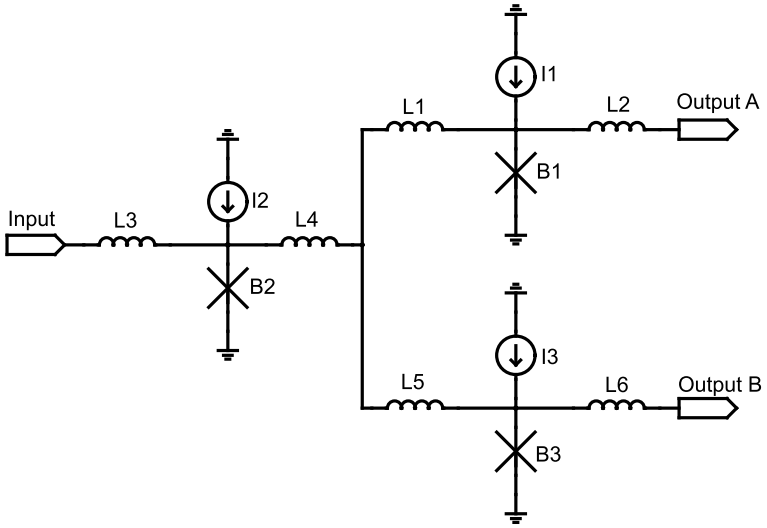


Figure 2.8: Circuit schematic of an RSFQ Pulse Splitter

switch to its resistive state, therefore the pulse is allowed to be applied to the loop in order to release the trapped fluxon. Similarly, B_2 is used to protect against a subsequent pulse at $InputA$ in instances where a fluxon is already stored in the loop.

Another circuit of interest is the pulse splitter depicted in Figure 2.8. Shunt resistances are again omitted for the sake of clarity.

An arriving pulse once again switches B_2, allowing a fluxon to enter the circuit. Assuming that B_1 and B_3 is biased slightly below their critical current value, and that L_1, L_4 and L_5 are sufficiently small, the current associated with the arriving fluxon could be sufficient to switch both B_1 and B_3, thereby generating two SFQ pulses, one at each output. The junction can therefore be used to amplify the arrived fluxon, that was split between two junctions in order to generate two output pulses. At this point, the output pulses will each consist of a single fluxon.

Hence, using this fundamental behaviour, many different types of logical circuits can be constructed. The interested reader is pointed to [1] for more circuitry details.

Chapter 3

Monte Carlo Methods for Yield Estimation

3.1 Introduction

In the manufacturing of integrated electronic circuits, either semiconductor or superconducting circuits, some statistical variations are present that can influence the functionality or performance of a circuit. In the semiconductor industry, the variations of interest can include changes to the transistor construction, such as oxide thickness, doping value and geometric variations. In low temperature superconducting electronics, the variations usually entail the geometry of the Josephson junctions, that is, the junction area and the geometry of other passive elements, such as resistors and inductors [13].

The percentage of circuits created that are functional and fall within the accepted performance bound is called the circuit yield. Manufacturers of such integrated circuits normally gather a large amount of practical data after which a statistical analysis is performed to ascertain a distribution and variance for each of the parameters. In general the distributions tend to be Gaussian with 3-sigma or 6-sigma information rendered as a variance measure.

It is the responsibility of the designer to maximise the yield of the designed circuit by using various optimisation algorithms and techniques, some of which are investigated in Chapter 6 of this document. Common to each of these algorithms, though, is the method employed to obtain the yield of the circuit given the current circuit parameters.

Analytically the circuit yield can be calculated as follows: Assuming that the circuit parameters under investigation form the random vector Y, each with its respective distribution (possibly Gaussian), so as to create a joint distribution function $f(\cdot)$, the circuit yield will then be the multi-dimensional integral of the joint distribution function across the acceptance region Ω_a. This equates to the expected value of the joint distribution function and can theoretically be stated as follows:

$$Y = \int_{\Omega_a} f(\mathbf{X}) dx \tag{3.1.1}$$

The challenge that designers face is that no analytical mapping generally exists between selected circuit parameters and the acceptance region Ω_a. Designers are therefore compelled to make use of simulations, which are usually expensive in terms of time when testing and measuring the functionality and performance of a circuit under test. A Monte Carlo method is therefore used to serve as an *unbiased estimator* of equation 3.1.1.

The Monte Carlo method is defined as follows:

$$\overline{Y} = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{X}) \tag{3.1.2}$$

where N refers to the number of samples drawn from the joint probability distribution. Due to the law of large numbers [14], the approximation converges to the result presented in equation 3.1.1 with a probability of 1 in the limit where $N \to \infty$.

It is valuable to define a figure of merit for an approximation algorithm as this allows one to gain confidence in the algorithm itself as well as to compare it with other approximation algorithms. For unbiased approximates such as Monte Carlo, the variance of the solution can be used as a figure of merit [14]. The equation to calculate the associated variance is presented in equation 3.1.3.

$$\sigma_s^2 = \mathcal{E}[Y - \overline{Y}]^2 \tag{3.1.3}$$

In effect, $\overline{Y}$ is in itself a random variable since multiple runs using the same number of samples (Equation 3.1.2) will result in different approximations. The use of the central limit theorem [14, 15] demonstrates that the variance in equation 3.1.3 decreases at a rate of $\mathcal{O}(N^{-\frac{1}{2}})$ which implies that four times as many samples are required from the distribution in order to decrease the variance by a factor of 2 (and therefore the error of the approximation method). The aim of investigating various Monte Carlo methods is to find an algorithm that decreases its error faster than the Crude Monte Carlo (CMC) approximation presented in equation 3.1.2.

In this chapter, two Monte Carlo methods, commonly used in various engineering and financial fields are investigated for their applicability to RSFQ circuit yield estimation. The first method, Latin Hypercube Sampling, makes use of sample space stratification while the second, Sobol sequences, uses a low discrepancy sequence to lower the variance of the approximation. The aim of both of these methods is to draw samples from the joint distribution function in a more orderly fashion than a pseudo-random number generator, generally used with CMC, would be able to. In the following two sections, Latin Hypercube Sampling and Sobol sequences are explained in detail, followed by a practical example of each of the two methods.

3.2 Latin Hypercube Sampling

Stratified sampling constitutes a method of sub-dividing a sample space into disjoint regions called strata. Samples are then drawn according to the conditional probability of each strata. Therefore, assuming that one can sub-divide the joint distribution function into m distinct strata, each of which is selected with the probability p_i where $i = [1...m]$, this selection will then represent a random variable which can be called Z. With the use of the tower property of expected values, the stratified expected value of equation 3.1.1 can be written as follows [15]:

$$Y = \mathcal{E}\mathcal{E}[Y|Z] = \sum_{i=1}^{m} p_i \mathcal{E}[Y|Z = i] \tag{3.2.1}$$

Subsequently, the expected value of T can be calculated by using the expected values of the conditional distributions of the strata. Stratification therefore ensures that the samples are more evenly spaced than those generated using Crude Monte Carlo sampling from equation 3.1.2.

However, although very useful, stratified sampling tends to become too computationally intensive for large dimensional sample spaces while the calculation of the cumulative distribution functions can become cumbersome. Hence another method, Latin Hypercube Sampling, was proposed by [16] to circumvent this dimensional explosion.

Latin Hypercube Sampling modifies the stratified sampling method by dividing each dimensional range into regions of equal marginal probability. A single point is then sampled from each of these strata. Finally, these points are randomly matched between dimensions to form samples for investigation.

For example, given two random variables, both of which are described by normal distributions forming a joint probability function from which samples could be drawn, the first step would be to stratify the marginal probabilities of each random variable. Supposing further that 8 strata per marginal probability are selected by using a uniform distribution $\mathcal{U}$ between 0 and 1 as the starting point, this would result in each strata having an equal probability of $p_i = 0.125$. A random sample is then drawn from each of the distributions associated with the strata. A possible sampling is depicted in Table 3.1

VAR_{1U} in the table are the random, uniformly distributed, samples drawn from each strata of the marginal distribution of the first random variable. It is evident that the samples are generally equally spaced. $Var1_G$ is the equivalent Gaussian sample drawn by using the inverse transform method with a mean of 0 and a standard deviation of 1. VAR_{2U} and VAR_{2G} are the equivalent samples for the second random variable that were drawn by using the same Gaussian distribution parameters.

Figure 3.1 graphically illustrates the sampling of the first random variable. The figure also indicates the cumulative distribution function of a Gaussian

VAR_{1U}	VAR_{1G}	VAR_{2U}	VAR_{2G}
0.1	-1.28155	0.08	-1.40507
0.15	-1.03643	0.2	-0.841621
0.29	-0.553385	0.32	-0.467699
0.45	-0.125661	0.43	-0.176374
0.6	0.253347	0.58	0.201893
0.69	0.49585	0.65	0.38532
0.8	0.841621	0.83	0.954165
0.91	1.34076	0.96	1.75069

Table 3.1: Random Samples across Two Gaussian Random Variables

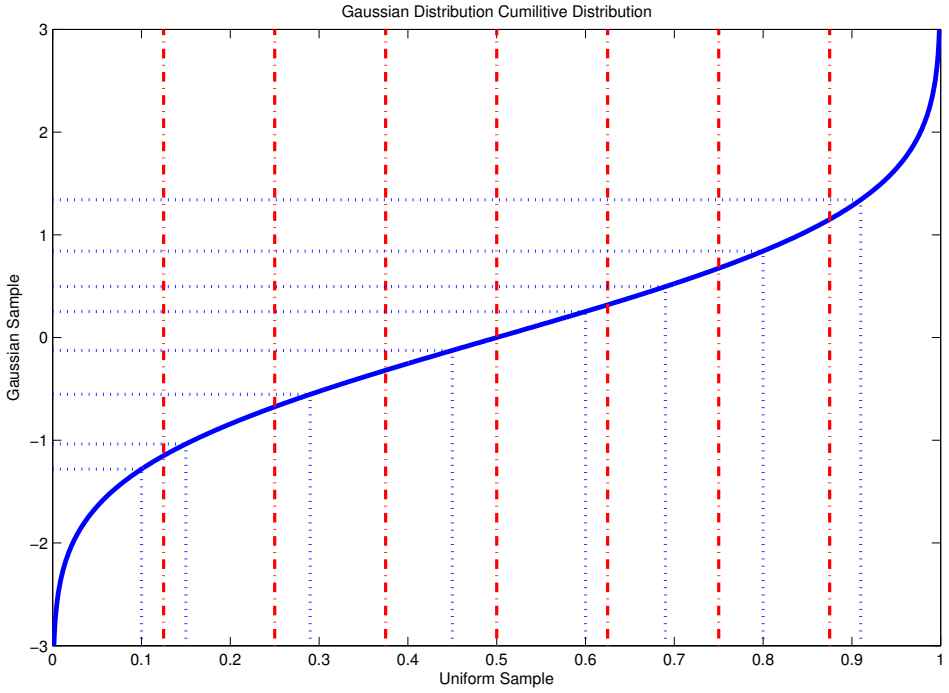


Figure 3.1: Obtaining Latin Hypercube samples from a Gaussian distribution.

distribution with a mean of 0 and a standard deviation of 1. The x-axis in Figure 3.1 indicates the probability that a sampled number from the distribution could fall below the respective y-axis value of the cumulative distribution graph. The x-axis is graphically stratified and a random sample represented by the vertical lines within each strata, is drawn. The corresponding Gaussian distributed value for each sample is then calculated (shown as the horizontal lines) and stored for later use.

Finally, random permutations of the columns VAR_{1G} and VAR_{2G} are selected without replacement in order to generate the sample points for the two

dimensions of the sample space.

A possible selection is depicted in Figure 3.2 where the horizontal and vertical lines represent the strata limits and the points indicate the sample point selections.

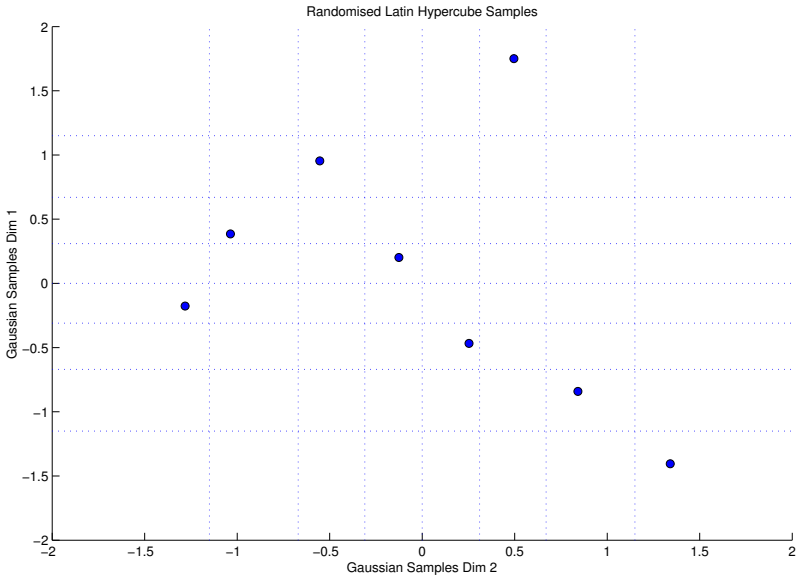


Figure 3.2: Example of Latin Hypercube sampling of a Gaussian random variable.

An example of the LHS method employed with the RSFQ circuit yield approximation is presented in Section 4 of this chapter.

3.3 Sobol Sequences

Another sampling method frequently employed with Monte Carlo simulations is quasi-random numbers. These sequences are in fact not random at all but are rather deterministically generated and exhibit the kind of behaviour of being generally more evenly spaced through the sample space than are independent and identically distributed values. Mathematically, the sequence is said to possess a low discrepancy [17]. For instance, if $\mathscr{E}$ is a collection of subsets of $[0,1)^d$, where d is the dimension of the normalised sample space, and $\mathscr{P} = \{u_1, ..., u_N\}$ is a set of points in $[0,1)^d$ the d-dimensional unit cube, then the star-discrepancy of the set of points can be stated as:

$$D_{\mathscr{E}}(\mathscr{P}_N) = \sup_{C \in \mathscr{E}} \left| \frac{1}{N} \sum_{i=1}^{N} I_{u_i \in C} - \int I_{u \in C} du \right| \tag{3.3.1}$$

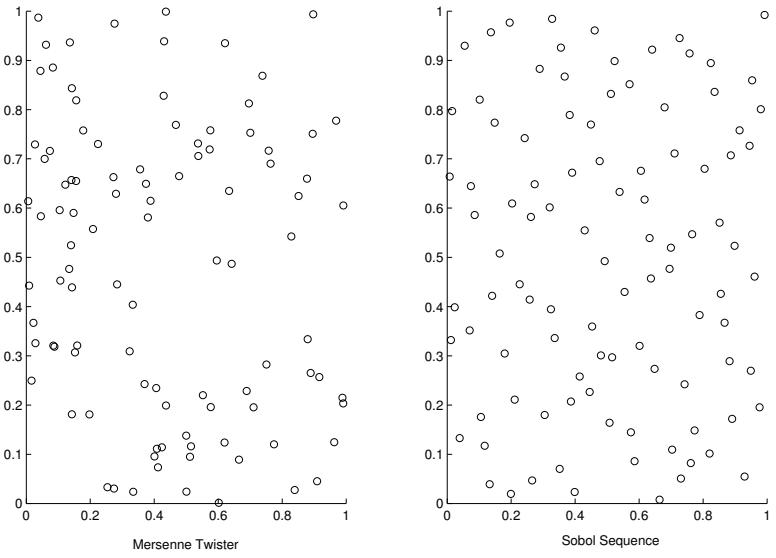


Figure 3.3: Illustration of pseudo-random vs quasi-random sampling.

where sup denotes the *supremem* operator, the sum term is the proportion of points in C, and the integral is the volume of C in proportion to the unit cube volume. The measure of low discrepancy in equation 3.3.1 returns the worst-case (highest) discrepancy value of the sets of points. The discrepancy returns the worst error if the sum estimate is used to estimate the volume of the sets of points when the sample space is divided into volumes of the same size. Too few or too many points in a sample volume would cause a significant error and therefore also a significant discrepancy. Hence it follows that a low discrepancy sequence is generally equi-distributed, that is, a volume in a low discrepancy sequence contains a number of points proportional to the volume size. Figure 3.3 graphically illustrates the difference between pseudo-random numbers and low discrepancy sequences. The sample points on the left of the figure were generated with the use of the Mersenne Twister method (a popular pseudo number generator available in the Boost C++ library [18]) while those on the right were generated with the use of a Sobol sequence. It is noticeable that the quasi-random sequence tends to be more uniformly spaced throughout the sample space than the pseudo-random samples.

The Sobol sequence is a digital sequence proposed in [19]. Although only the generation of the sequence according to [20] is described, the interested reader is referred to [19] for the theoretical underpinnings of the sequence.

The first step is to freely select a primitive polynomial of degree d in $\mathcal{Z}_2 = \{0, 1\}$ in order to generate a one-dimensional Sobol sequence as is indicated in equation 3.3.2.

$$x^d + a_1 x^{d-1} + ... + a_{d-1} + 1 \tag{3.3.2}$$

Next, a sequence of positive integers is defined by using the following recurrence relation for the k_{th} number in the sequence:

$$m_k = 2a_1 m_{k-1} \oplus 2^2 a_2 m_{k-2} \oplus 2^{d-1} a_{d-1} m_{k-d+1} \oplus 2^d m_{k-d} \oplus m_{k-d} \tag{3.3.3}$$

The recurrence relation in equation 3.3.3 requires initial values m_k with $k = [1..d]$ which can be freely selected as long as the values are odd and less than 2^k. The initial values for equation 3.3.3 and the primitive polynomials can be obtained by using the algorithms described in [21] which in turn were used to publish the values obtainable from [22].

Direction numbers, presented in equation 3.3.4 are then generated and used to create the final sequence:

$$v_k = \frac{m_k}{2^k} \tag{3.3.4}$$

The ith point in the sequence can subsequently be generated as follows:

$$x_i = i_1 v_1 \oplus i_2 v_2 \oplus ... \tag{3.3.5}$$

where i_k is the k_{th} digit from the right when i is written in binary form.

For example, commencing with a primitive polynomial of degree 4 with $a_1 = 0$, $a_2 = 0$ and $a_3 = 1$ will result in the primitive polynomial presented in equation 3.3.6.

$$x^4 + x^3 + 1 \tag{3.3.6}$$

Commencing with the initial direction numbers $m_1 = 1$, $m_2 = 3$, $m_3 = 7$, $m_4 = 5$, the next six values can be generated by using equation 3.3.3 as demonstrated below where all m_k are presented in their binary format:

$$\begin{aligned} m_5 &= 2^3 m_2 \oplus 2^4 m_1 \oplus m_1 \\ &= 2^3 (0011)_2 \oplus 2^4 (0001)_2 \oplus (0001)_2 \\ &= (1000)_2 \oplus (0000)_2 \oplus (0001)_2 \\ &= (1001)_2 \\ &= 9 \end{aligned} \tag{3.3.7}$$

Likewise, m_6 to m_{10} can also be calculated as observed from the results in Table 3.2.

Subsequently, with the use of equation 3.3.5, the numbers of the sequence can be calculated for x_5 as demonstrated in equation 3.3.8.

m_k	v_k
$(0001)_2$	$(0.1)_2$
$(0011)_2$	$(0.11)_2$
$(0111)_2$	$(0.111)_2$
$(0101)_2$	$(0.0101)_2$
$(1001)_2$	$(0.01001)_2$
$(1011)_2$	$(0.001011)_2$
$(1111)_2$	$(0.0001111)_2$
$(1101)_2$	$(0.00001101)_2$
$(0001)_2$	$(0.000000001)_2$
$(0011)_2$	$(0.0000000011)_2$

Table 3.2: Sobol sequence calculation: values for m_k and v_k in binary.

$$\begin{aligned} x_5 &= i_1 v_1 \oplus i_2 v_2 \oplus i_3 v_3 ... \\ &= (1)v_1 \oplus (0)v_2 \oplus (1)v_3 \\ &= (0.1)_2 \oplus (0.111)_2 \\ &= (0.011)_2 \\ &= 0.3750 \end{aligned} \tag{3.3.8}$$

Values for the first 10 elements of the sequence are listed in Table 3.3.

i	x_i
1	$(0.1)_2 = 0.5$
2	$(0.11)_2 = 0.75$
3	$(0.01)_2 = 0.25$
4	$(0.111)_2 = 0.875$
5	$(0.011)_2 = 0.375$
6	$(0.001)_2 = 0.125$
7	$(0.101)_2 = 0.625$
8	$(0.0101)_2 = 0.3125$
9	$(0.1101)_2 = 0.8125$
10	$(0.1001)_2 = 0.5625$

Table 3.3: Values for x

3.4 Practical Example

An AND gate and an XOR gate were selected to investigate the effects of the Monte Carlo methods on the yield calculation of RSFQ circuits. The AND

gate is presented in Figure 3.4 with its component values listed in Table 3.4 while the XOR gate is presented in Figure 3.5 with its components listed in Table 3.5. For both circuits the shunt resistances and parasitic elements were omitted for the sake of clarity. All input pulses are applied through a DC-SFQ circuit connected to the circuit under test using a Josephson Transmission Line while the outputs are terminated using a 2Ω resistor connected to the circuit also through a Josephson Transmission Line.

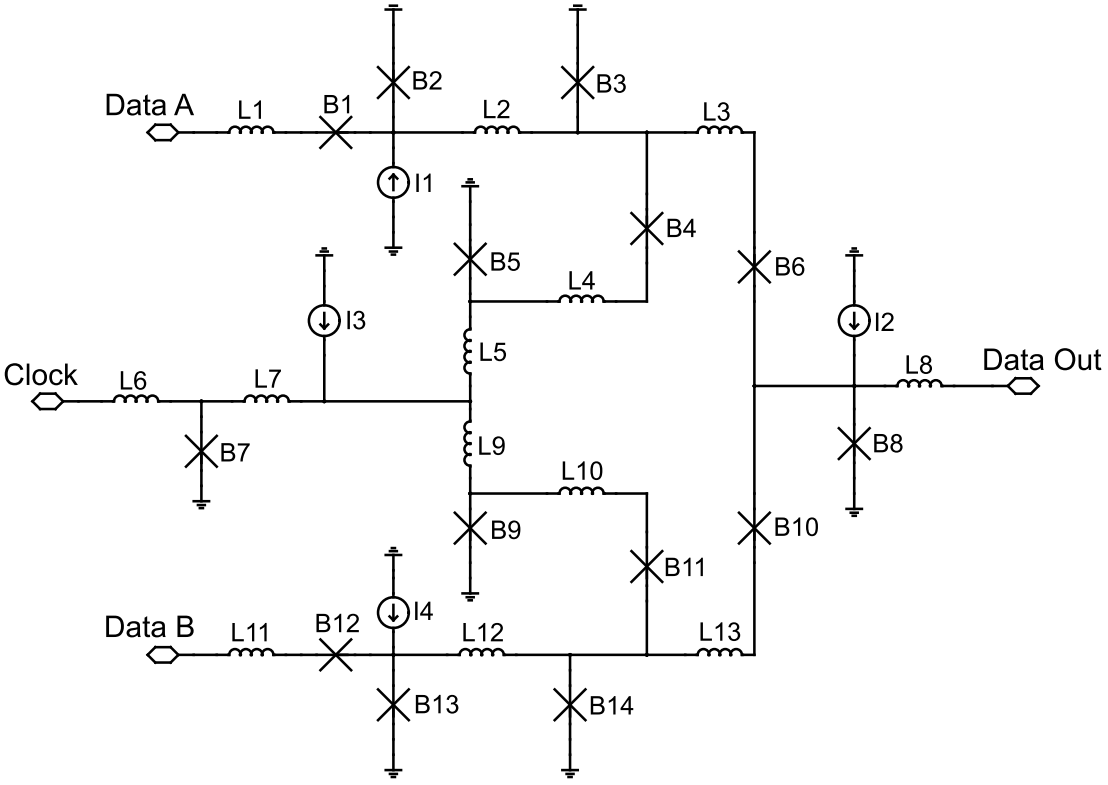


Figure 3.4: Circuit schematic of an un-optimised RSFQ AND gate.

The AND gate consists of two fluxon storage loops B_2—L_2—B_3 and B_{13}—L_{12}—B_{14}. SFQ pulses entering *DataA* and *DataB* are stored in these loops respectively. A *Clock* input is used to release both of these stored fluxons simultaneously, which changes the bias currents of B_6 and B_{10} so that B_8 switches before B_6 and can remove the SFQ pulses from the circuit. This causes an output pulse to be generated. If only one fluxon is stored, a *Clock* pulse will cause either B_6 or B_{10} to switch before B_8, thereby throwing the pulse out of the circuit without generating an output. A pulse splitter is formed through B_5 and B_9 to apply the input clock to the fluxon storage loops. B_1 and B_{12} serve as series junctions to remove any input data pulses when a respective fluxon is already stored while B_4 and B_{11} are used to remove the *Clock* pulses if no fluxon is stored in the storage loops.

Junctions	Inductors	Bias Currents
$B_1 : 200\,\mu m^2$	$L_1 : 2.47\,pH$	$I_1 : 187\,\mu A$
$B_2 : 250\,\mu m^2$	$L_2 : 9.4\,pH$	$I_2 : 256\,\mu A$
$B_3 : 325\,\mu m^2$	$L_3 : 2.75\,pH$	$I_3 : 460\,\mu A$
$B_4 : 275\,\mu m^2$	$L_4 : 3.01\,pH$	$I_4 : 187\,\mu A$
$B_5 : 175\,\mu m^2$	$L_5 : 0.69\,pH$	
$B_6 : 150\,\mu m^2$	$L_6 : 1.25\,pH$	
$B_7 : 300\,\mu m^2$	$L_7 : 0.81\,pH$	
$B_8 : 350\,\mu m^2$	$L_8 : 1.1\,pH$	
$B_9 : 175\,\mu m^2$	$L_9 : 0.69\,pH$	
$B_{10} : 150\,\mu m^2$	$L_{10} : 3.01\,pH$	
$B_{11} : 275\,\mu m^2$	$L_{11} : 2.47\,pH$	
$B_{12} : 200\,\mu m^2$	$L_{12} : 9.4\,pH$	
$B_{13} : 250\,\mu m^2$	$L_{13} : 2.75\,pH$	
$B_{14} : 325\,\mu m^2$		

Table 3.4: Un-optimised RSFQ AND gate circuit parameters.

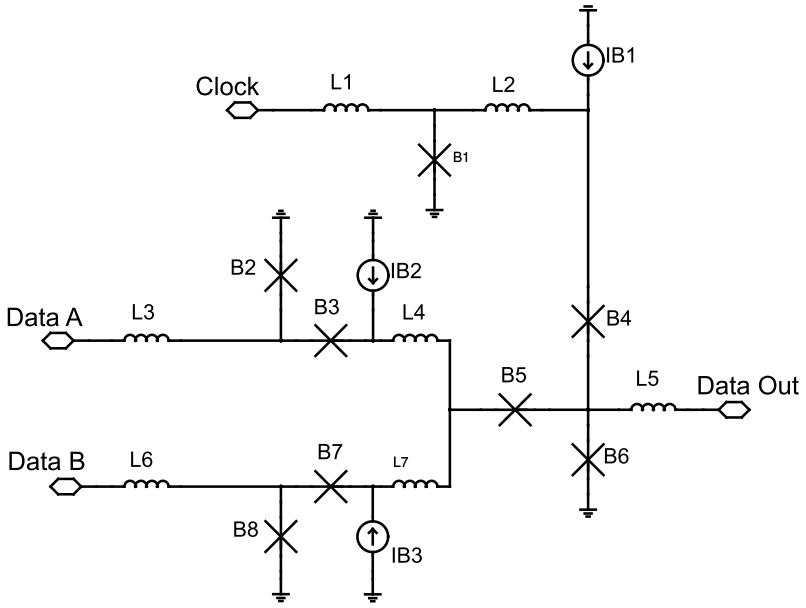


Figure 3.5: Circuit schematic of an un-optimised RSFQ XOR Gate.

The XOR gate consists of a single storage loop which is constructed from different components depending on which data input pulse arrived first. The loop consists of B_2—B_3—L_4—B_5—B_6 if a pulse on $DataA$ arrives first and B_8—B_7—L_7—B_5—B_6 if a pulse on $DataB$ arrives first. Junctions B_3 and B_7 serve as escape junctions for the non-receiving input in order to ensure that no pulses are propagated backwards through the system. If a pulse has already arrived, by means of $DataA$ for example, a follow-up pulse on $DataB$ will

Junctions	Inductors	Bias Currents
$B_1 : 150\,\mu m^2$	$L_1 : 2.48\,pH$	$I_1 : 100\,\mu A$
$B_2 : 150\,\mu m^2$	$L_2 : 1\,pH$	$I_2 : 100\,\mu A$
$B_3 : 150\,\mu m^2$	$L_3 : 2.48\,pH$	$I_3 : 100\,\mu A$
$B_4 : 150\,\mu m^2$	$L_4 : 4.12\,pH$	
$B_5 : 200\,\mu m^2$	$L_5 : 4.05\,pH$	
$B_6 : 200\,\mu m^2$	$L_6 : 2.48\,pH$	
$B_7 : 150\,\mu m^2$	$L_7 : 4.12\,pH$	
$B_8 : 150\,\mu m^2$		

Table 3.5: Un-optimised RSFQ XOR gate circuit parameters.

cause B_5 to switch thereby releasing the already stored fluxon. Consequently, the XOR logical operation is implemented. The arrival of a *Clock* pulse will release a stored fluxon through B_6, generating an output pulse. If no fluxon is stored, B_4 is used as the escape junction for the *Clock* pulse.

An automated state machine extraction method, which is discussed in detail in Chapter 5 of this document, was used to test the functionality of the AND and XOR gates. This method tests a perturbed circuit (such as is employed in yield calculations) against the nominal state machine representation of the circuit. No performance measurement (latency, etc.) was, however, carried out during the functionality testing.

To simplify the investigation, the local and global process variations of the relevant parameters described in [13] were reduced to a single normal distribution for each parameter, each with a standard deviation of 0.13. As a benchmark, a Crude Monte Carlo run was executed with 20000 samples which returned a yield value of 62.580% for the AND gate with given parameters and variations. The equivalent yield for the XOR gate after 20000 was 43.55%.

A robust approach for comparing the three methods is to conduct a comparison of how the variance reduces for each method versus the number of samples drawn. Nine sample steps were investigated, commencing with 100 samples and ending with 500 samples, with each step adding 50 more samples to the investigation. For each sample step, each algorithm was run 100 times in order to quantify the variance. The variance of each method for each sample point could be calculated, by means of equation 3.1.3 and using a yield value of 62.580% for the AND gate and 43.55% for the XOR gate.

In order to investigate the variance of a Sobol sequence some element of randomisation needs to be added. This is due to the Sobol sequence being deterministically defined. In effect, given the same primitive polynomial and initial direction vectors, the same sequence will always be generated. Hence, a simple shuffle method was used for randomisation. All the Sobol sequence points were pre-created after which each parameter sequence was shuffled by employing a pseudo-random number generator. For these tests a Mersenne Twister pseudo-random number generator, available in the C++ boost library

[18], was applied. Various other randomisation methods are also available and should be investigated in future studies [14, 15].

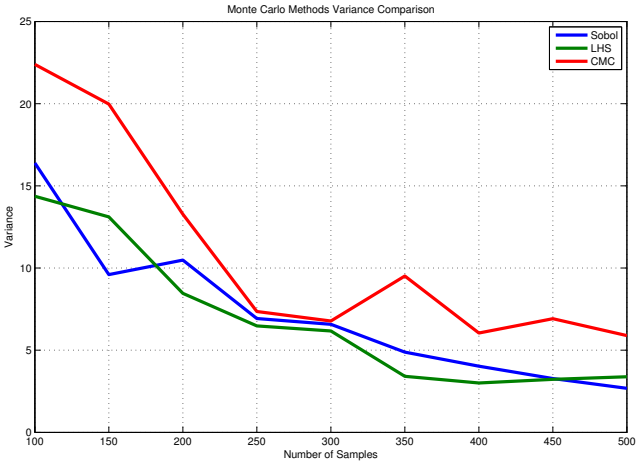


Figure 3.6: Monte Carlo method variance comparison for an RSFQ AND Gate.

The results of the variance comparison for the RSFQ AND gate are presented in Figure 3.6. It is evident that both the Latin Hypercube and the randomised Sobol methods tend to perform better than the Crude Monte Carlo method. Some assumptions can be made due to the fact that Latin Hypercube Sampling performs better than Crude Monte Carlo sampling. As discussed in [14], Latin Hypercube Sampling tends to only perform better than Crude Monte Carlo sampling when the function to be sampled contains some parameters which affect the variance independently of the other parameters. In effect, the sampled function can be reduced to a summation of functions that vary the yield over one dimension and functions that vary over more than one dimension. Future work should therefore include an investigation into the effective dimensionality of RSFQ circuits. For semiconductor circuits the effective dimensionality tends to be substantially lower than the number of parameters [14] which leads to substantial speed-ups for circuit modelling. The same might be true for RSFQ circuits.

Sobol sequences also tend to reduce the variance of functions which contain one-dimensional components [14]. It was also mentioned that Sobol sequences have a lower discrepancy for the lower ($l < 10$) dimensions. It would therefore be preferable to use the lower dimensional Sobol points for the more sensitive parameters in the circuit. An interesting future test would be to initiate a margin analysis on the components of an RSFQ circuit and explicitly use the lower dimensional Sobol points for these parameters.

To appreciate the advantages suggested in Figure 3.6, one can calculate the confidence interval as indicated in equation 3.4.1:

$$(\overline{X} - z_{1-a/2}\frac{\sigma}{\sqrt{N}}, \overline{X} + z_{1-a/2}\frac{\sigma}{\sqrt{N}}) \tag{3.4.1}$$

The equation is governed by the estimated standard error [15], $\frac{\sigma}{\sqrt{N}}$. When one examines the variance samples at 500 runs in Figure 3.6, it is evident that the values 5.888, 3.382 and 2.682 for CMC, Latin Hypercube and Sobol respectively are obtained. This leads to estimated standard errors of 0.108, 0.082 and 0.073 respectively. In practical terms, to achieve the same level of confidence in the yield approximation, one would require 57% of the number of runs needed by CMC when using Latin Hypercube and 45% of that needed by CMC for the Sobol sequence.

The results of the XOR gate are depicted in Figure 3.7.

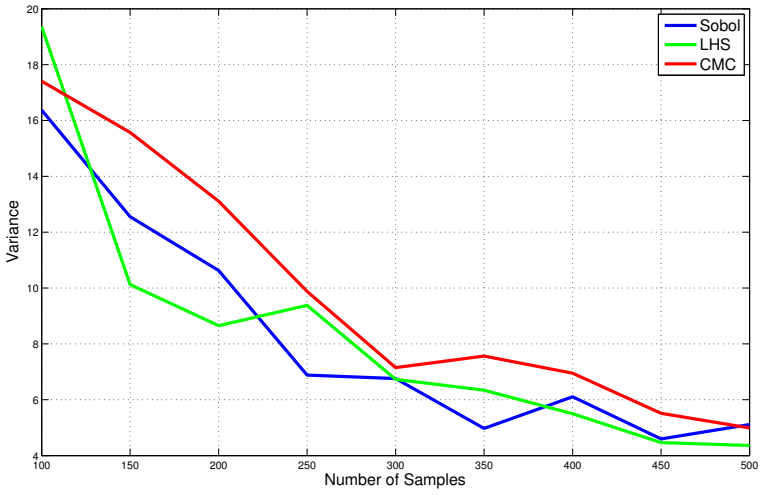


Figure 3.7: Monte Carlo Method Variance Comparison of RSFQ XOR Gate

Once again it is evident that both the Latin Hypercube sampling and the Sobol sequence provides a substantial reduction in variance. The difference in the XOR gate is more evident when a lower number of samples are drawn. The values of the samples at 450, which subjectively seems to be following the trend in the figure most closely, are 5.5126, 4.4662 and 4.597 for the CMC, Latin Hypercube and Sobol sequences respectively, which lead to standard error values of 0.107, 0.0996 and 0.101. This provides the same confidence interval in 81% of the runs for the CMC samples required by Latin Hypercube sampling and in 83% of the runs for the CMC samples required by the Sobol sequence.

Considering Figure 3.6 and Figure 3.7, it is evident that Latin Hypercube sampling as well as Sobol sequences can provide a measurable variance reduction when used in conjunction with RSFQ yield approximations. However, the

noise in the data samples does encumber the calculation of a numeric value for the variance reduction. Future work should therefore include more than 100 yield approximations per number of samples so that a more accurate numerical value of the variance reduction can be calculated.

Chapter 4

Artificial Neural Networks

4.1 Introduction

Another method for increasing the speed of yield calculations is to employ a function approximator. One of the most celebrated, and most researched, approximators is that of artificial neural networks(ANN). These networks make use of a very basic model of the neural connections in the human brain, along with training data, to iteratively learn to approximate a given function. If a neural network can be trained to approximate the yield search space accurately by using a limited number of Monte Carlo samples, any future required samples can be gathered through subsequent runs of the network. This could save a significant amount of time for optimisation algorithms that depend on multiple Monte Carlo runs. Artificial neural networks occupy a vast field of research. Therefore only the most documented network, the multi-layer Perceptron, was investigated regarding yield estimation. In addition, numerous research has also been conducted on yield estimation via artificial neural networks in the semiconductor industry [23, 24, 25, 26]. However, to the best knowledge of the author this technique has not yet been applied for yield estimation of RSFQ circuits.

Section 2 of this chapter encompasses the fundamentals of artificial neural networks, specifically regarding multi-layer Perceptron networks. Section 3 then applies the artificial neural network to an RSFQ digital logic circuit to ascertain whether it can serve as a viable approximator.

4.2 Artificial Neural Network Fundamentals

A simplified description of a biological neuron pertaining to artificial neural networks can be found in [27]. The neuron consists of a cell body, containing the nucleus and is surrounded by dendrites that receive stimuli from other neurons. Stimuli leave the neuron through the axon, which connects the neuron to downstream neurons. Stimuli are transferred by means of an electric charge

exchange created by the diffusion of ions. However, not all of the interconnections entering the neuron are equally weighted as some of them have priority over others. Interconnections can also inhibit the transmission of stimuli.

4.2.1 The Building Blocks of Artificial Neural Networks

Most of the artificial neural networks investigated currently are based on a biological model proposed by [28]. An illustration of the proposed model can be seen in Figure 4.1 [27].

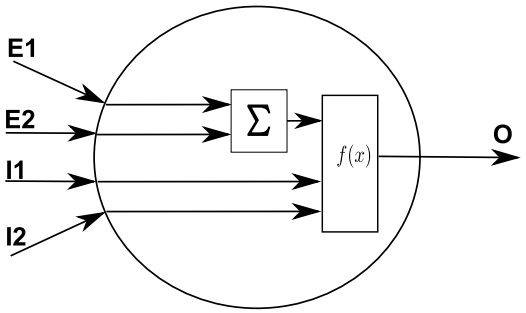


Figure 4.1: The McCulloch-Pitts neuron model.

In this figure, inputs designated by an E are excitation inputs and inputs designated by an I are inhibitory inputs. Furthermore, a threshold value T is associated with the neuron. If the number of the active excitation inputs is higher than the threshold value at any one time, the output of the neuron will be active. However, if the threshold value is not reached or if any of the inhibitory inputs are active, the neuron output will be inactive. It was also demonstrated that basic logical operations such as AND, OR and NOT could be implemented using the proposed model.

Further refinements to the neuron model were proposed by [29, 30, 31], eventually leading to the ADALINE (Adaptive Linear Neuron and later Adaptive Liner Elements) and the Perceptron, whose elements are shown in Figure 4.2.

Structurally there is very little difference between the ADALINE and the Perceptron. Both eventually contained the improvements of replacing the excitation and inhibitory inputs with a numerical value (1 and 0 or -1 and 1). Both of these modelling methods also investigated the influence of different threshold functions. Each of the inputs were also associated with a weight value to indicate the relative priority of the input. A bias term, that is, an input whose value is fixed at 1 (or -1) was also added to shift the effective threshold value. The transfer function of such an element for a Signum threshold function is presented in equation 4.2.1.

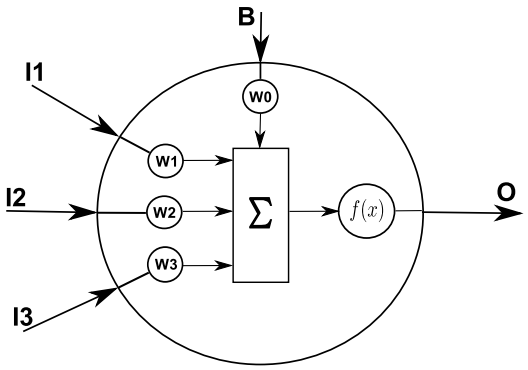


Figure 4.2: ADALINE and Perceptron elements.

$$O = 1 \quad when \sum_{k=i}^{n} i_k w_k + w_0 > T$$
$$= -1 \quad otherwise \tag{4.2.1}$$

It is evident from equation 4.2.1, that each element forms a *linear discriminator*. The sample space of these elements consist of two regions, which are divided by a linear boundary. For example, in the sample space in Figure 4.3, a discriminator implementing the AND function is presented where the output is 1 above and -1 below the division line.

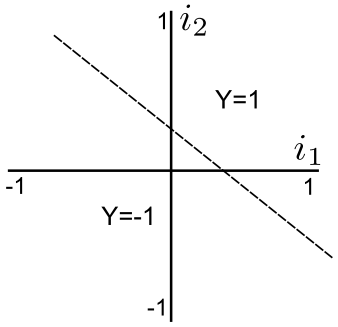


Figure 4.3: Single element sample space of the AND function.

A possible implementation of the discriminator would thus be a two input element with a Signum threshold function, weights $w_1 = 1$, $w_2 = 1$ and a bias weight of $w_0 = 0.5$ when the bias input is -1. When the inputs are both 1, the sum term plus the bias of equation 4.2.1 is 1.5 which causes the output to be active. Any other input combination would cause the output to be suppressed.

The limitation of the single Perceptron or ADALINE (or a single layer for that matter) is that these elements can only represent *linearly separable* functions. An example of this limitation is illustrated in Figure 4.4.

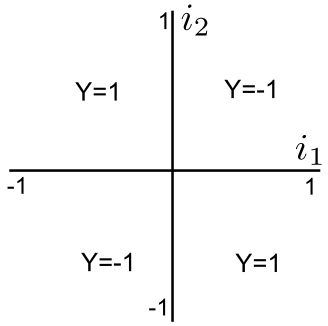


Figure 4.4: Single element sample space of the XOR function.

In this figure, the requisite sample space configuration is depicted in order to implement an XOR function. However, it is evident that it is impossible to draw a straight line on the figure to separate the portions where the output should be 1 from the portions where the output should be -1. The XOR function is therefore not implementable with a single Perceptron/ADALINE element or with a single layer of the elements.

The solution to the linear separation limitation as discussed in [27, 32, 33] is therefore to create multiple layers of Perceptron/ADALINE elements connected in a feed-forward topology as illustrated in Figure 4.5.

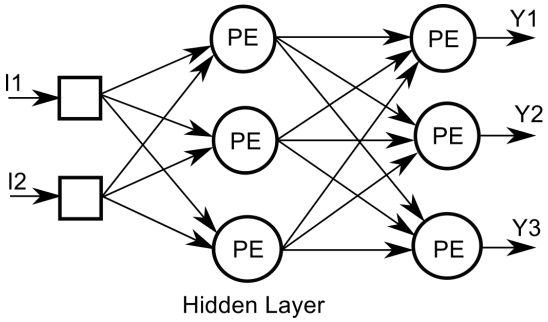


Figure 4.5: Example of a multi-layer ANN topology.

In this topology, any layer not connected to an output is called a hidden layer. Also notable is the fact that information always propagates forward to the next layer in the network, hence the designation feed-forward network. The processing elements (PE) shown in the figure can either be Perceptron elements or ADALINE elements. As [32] indicates, a network with only three layers is sufficient to approximate any input-output mapping given a finite number of processing elements in each layer, a sigmoid threshold function (to be explained below) and successful training.

4.2.2 Training through the Back-Propagation method

The next step under investigation is the training of the network itself. Training of artificial neural networks can be divided into two categories: supervised or unsupervised. However, in this study, only supervised training is investigated for yield estimation purposes. The aim of the supervised training algorithm is to supply the artificial neural network with training data consisting of input values with expected output values. The network should then adjust the weights of the system (the free variables) to minimise an error function. One of the major differences between the Perceptron and the ADALINE is the error function definition.

The Perceptron makes use of equation 4.2.2 for error calculations.

$$\mathcal{E}(w) = \sum_{d} (t^d - o^d)^2 \tag{4.2.2}$$

The error is therefore the squared sum of the differences between the current output and the desired output of the Perceptron. The ADALINE, on the other hand, uses equation 4.2.3 for its error function.

$$\mathcal{E}(w) = \sum_{d} (t^d - \sum_{j=0}^{n} w_j i_j^d)^2 \tag{4.2.3}$$

In effect, the ADALINE error function is the sum of squared differences between the desired response and the sum of the weight-input pairs of the respective elements. The difference is therefore the "feedback" of the error function (before and after the threshold function). It is notable that the training algorithm (as shown below) employs a gradient method to adjust the weight parameters. This, however, causes a problem for equation 4.2.2 since if the Signum function is used as a threshold, the differential does not exist due to the discontinuous nature of the function. The solution to this dilemma was to employ a Sigmoid function, depicted in Figure 4.6, with the equation presented in 4.2.4. The derivative of the sigmoid function is presented in equation 4.2.5.

$$y = \frac{1}{1 + e^{-x}} \tag{4.2.4}$$

$$\frac{dy}{dx} = y(1 - y) \tag{4.2.5}$$

The training algorithm for the Perceptron network is called the back-propagation algorithm and is explained by means of the derivation found in [32]. The update rule for the input weights of the Perceptron uses the gradient of the error function (as presented in equation 4.2.2). The update rule is shown in equation 4.2.6:

$$\Delta w_i = -k \frac{\partial \mathcal{E}}{\partial w_i} \tag{4.2.6}$$

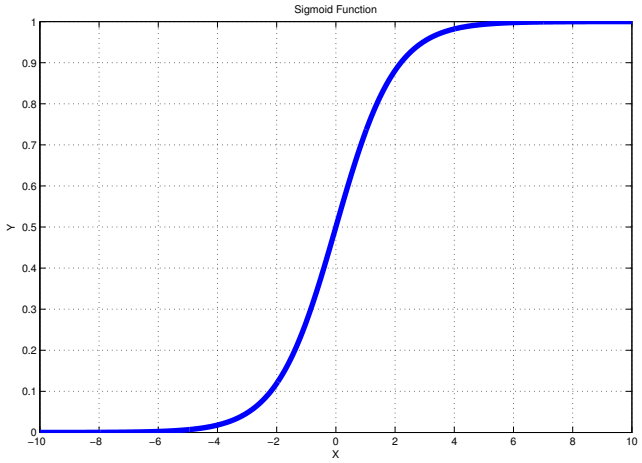


Figure 4.6: A Sigmoid function.

where k is the step size per iteration.

First the update equation of a single Perceptron and then the generalised equation of the back-propagation algorithm are derived. For the single Perceptron, x_i are the inputs, d_p is be the desired output and y_p is the actual Perceptron output across all the training data p.

Using equation 4.2.4, the output of a processing element (single Perceptron node) can be written as follows:

$$y_p = \frac{1}{1 + e^{-net_p}} \tag{4.2.7}$$

where

$$net_p = \sum_{i=0}^{n} (w_i x_i)_p \tag{4.2.8}$$

The summation is over all the inputs entering the respective processing element. Using $ep = (d_p - y_p)^2$ as the error of an individual Perceptron in conjunction with the above equations, the gradient function can be written as follows:

$$\begin{aligned}
\frac{\partial \mathcal{E}}{\partial w_i} &= \frac{1}{P}\sum_{p=1}^{P}\frac{\partial e_p}{\partial w_i} \\
&= \frac{1}{P}\sum_{p=1}^{P}\frac{\partial e_p}{\partial y_p}\times\frac{\partial y_p}{\partial w_i} \quad \text{(using the chain rule)} \\
&= \frac{1}{P}\sum_{p=1}^{P}-2(d_p-y_p)\times\frac{\partial y_p}{\partial w_i} \\
&= -\frac{2}{P}\sum_{p=1}^{P}(d_p-y_p)\times\frac{\partial y_p}{\partial net_p}\frac{\partial net_p}{\partial w_i} \quad \text{(using the chain rule)} \\
&= -\frac{2}{P}\sum_{p=1}^{P}(d_p-y_p)y_p(1-y_p)x_{ip} \quad \text{(using equation 4.2.5)} \qquad (4.2.9) \\
& \qquad (4.2.10)
\end{aligned}$$

Equation 4.2.6 therefore becomes:

$$\Delta w_i = \frac{2k}{P}\sum_{p=1}^{P}x_{ip}y_p(d_p-y_p)(1-y_p) \qquad (4.2.11)$$

Hence the back-propagation algorithm itself can be explained with a single example. Considering Figure 4.7, which consists of a three layer network, the first layer is merely used to propagate the inputs to all the processing elements.

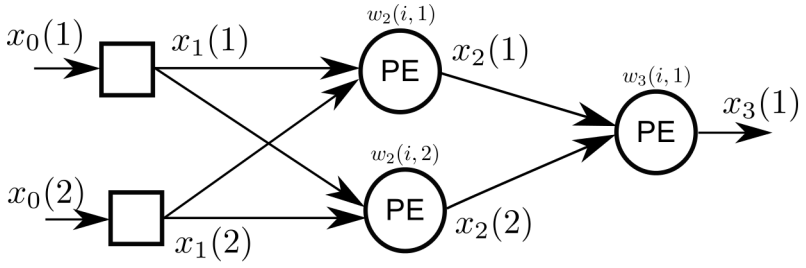


Figure 4.7: Multi-layer ANN topology as back-propagation example.

Each input (and output) has been named $x_l(i)$, where l is the layer number and i is the net number in the respective layer. The weights of each processing element are designated $w_l(i,k)$, where l is once again the layer, i is the input index and k is the processing element index in the layer. For the network illustrated here, the range of i is [0..2]. The $0th$ element of each weight is the weight associated with the bias input, which is not explicitly presented in the figure. For the sake of clarity the squared error will be used for a single training input instead of equation 4.2.2 which calculates the mean squared error over

all the training data. For example, the error function of the output Perceptron in Figure 4.7 is $\mathcal{E} = (d - x_3(1))^2$.

Supposing that the aim is to update weight $w_2(1,2)$, then a partial derivative of the error function with respect to $w_2(1,2)$ will be required in order to use equation 4.2.6. Furthermore, error derivatives can be expanded from $x_3(1)$ to $x_2(2)$ by using the chain rule as follows:

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_2(1,2)} &= \frac{\partial \mathcal{E}}{\partial x_3(1)} \times \frac{\partial x_3(1)}{\partial w_2(1,2)} \\ &= \frac{\partial \mathcal{E}}{\partial x_3(1)} \times \frac{\partial x_3(1)}{\partial x_2(2)} \times \frac{\partial x_2(2)}{\partial w_2(1,2)}\end{aligned} \tag{4.2.12}$$

Expanding each of the chain rule derivatives:

$$\frac{\partial \mathcal{E}}{\partial x_3(1)} = -2(d - x_3(1)) \quad \text{(from the squared error of the output)} \tag{4.2.13}$$

$$\frac{\partial x_3(1)}{\partial x_2(2)} = w_3(2,1)x_3(1)(1 - x_3(1)) \quad \text{using equation 4.2.5} \tag{4.2.14}$$

$$\frac{\partial x_2(2)}{\partial w_2(1,2)} = x_1(1)x_2(2)(1 - x_2(2)) \quad \text{using equation 4.2.5} \tag{4.2.15}$$

Substituting the above equations in equation 4.2.12 results in:

$$\frac{\partial \mathcal{E}}{\partial w_2(1,2)} = -2x_1(1)x_2(2)(1 - x_2(2))w_3(2,1)x_3(1)(1 - x_3(1))(d - x_3(1)) \tag{4.2.16}$$

which leads to the following update rule for $w_2(1,2)$:

$$\Delta w_2(1,2) = 2kx_1(1)x_2(2)(1 - x_2(2))w_3(2,1)x_3(1)(1 - x_3(1))(d - x_3(1)) \tag{4.2.17}$$

With the use of the aforementioned method the change in weight after each training run can now be calculated, starting from $w_3(i,1)$ to $w_2(i,j)$. Training can be achieved batch wise, where all the training data are used simultaneously to calculate the error function and update the weights, or incrementally, where the weights are updated after each individual run.

4.3 Practical Application

The ability to approximate the yield sample space by means of artificial neural networks was added to the software library that forms part of this work. The

artificial neural networks were implemented with the Fast Neural Network Library [34] by interfacing with the various yield calculation and functional testing functions of the software library.

The RSFQ AND and OR gates are investigated here in order to demonstrate the yield estimation ability of artificial neural network. The gates are again illustrated in figures 4.8 and 4.9 with the circuit parameters discussed in Chapter 3. All input pulses are applied through a DC-SFQ circuit connected to the circuit under test using a Josephson Transmission Line while the outputs are terminated using a 2Ω resistor connected to the circuit also through a Josephson Transmission Line.

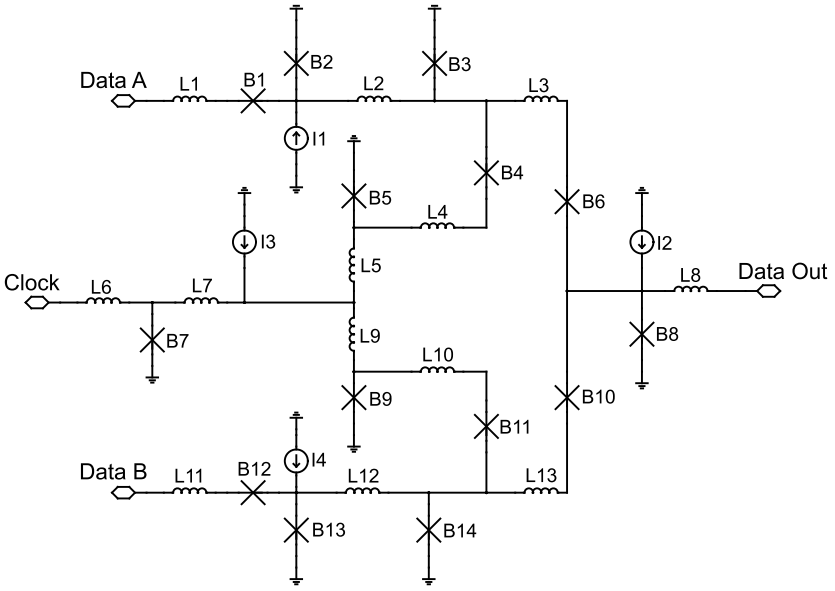


Figure 4.8: Circuit schematic of an un-optimised RSFQ AND Gate.

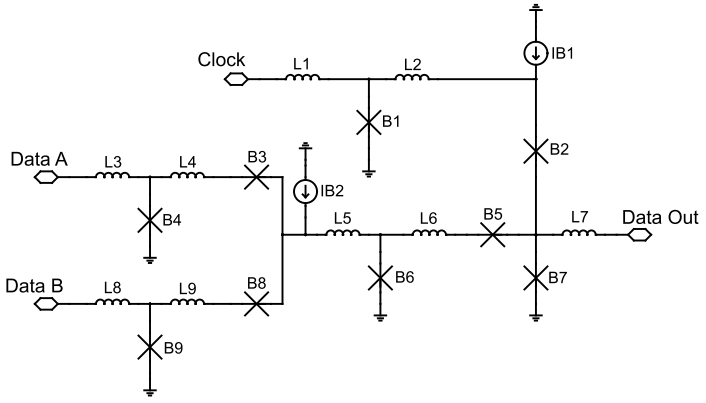


Figure 4.9: Circuit schematic of an un-optimised RSFQ OR Gate.

A neural network has many variables that can be modified and have an effect on its behaviour. Since it is very difficult to apply the theoretical framework of artificial neural networks to a specific problem, the variables are investigated empirically.

The first step in the creation of an artificial neural network is the generation of training data. This entails the sampling of the function to be approximated. In this case, it is the mapping of circuit parameters of the logical gate to a yield value. First a sampling strategy is selected and then a valid range of parameters is defined. The ranges of the components used with both circuits are listed in Table 4.1.

	L	B	I
Abs Min value	$0.5\,pH$	$100\,\mu m^2$	$100\,\mu A$
Abs Max value	$15\,pH$	$400\,\mu m^2$	$600\,\mu A$
Deviation	150 %	150 %	150 %

Table 4.1: Sample space definition for yield modelling.

Instead of using an absolute search space, each inductor, Josephson junction and current source was deviated by plus 150 % and minus 150 % of its nominal value to create the allowable range for each particular component. Absolute minimum and maximum values were also imposed on the deviated values to ensure that components with very large or very small nominal values remain in a practical implementable range.

An automated state machine extraction algorithm (which is discussed in Chapter 5) was used to test each deviated circuit for basic functionality. This entails extracting the state machine representation of the nominal circuit and testing it against the state machine representation of the deviated circuit. If functional, a Sobol sequence approximation (described in Chapter 3) with 500 runs was used to calculate the approximate yield. A standard deviation of 0.13 was used for all components when calculating circuit yield which was found to generally provide yield values between 40 % and 95 % for the circuits under test.

It is important to note that the AND gate consists of two identical arms, each of which are used to store and release a fluxon. In general, when designing such a circuit one would use the same parameter values for both of these arms. One would, for example refrain from using vastly different values for B_2 and B_{13}. In terms of the sample space approximation, and the optimisation algorithms described in Chapter 6, the user of the software library has the ability to define linked elements. The user would therefore inform the program that B_2 and B_{13} are linked which in turn would remove one free variable from the search space. This action has a dramatic effect on the performance of the algorithms. For the AND gate illustrated in Figure 4.8, linked elements were created between the two arms leading to 19 free variables in the circuit. A $\mathcal{B}_C$

value of 1 was used to calculate the shunt resistance of each junction. In an equivalent fashion, the two arms of the OR logical gate shown in Figure 4.9 can also be linked resulting in 16 free variables.

For the samples themselves a Latin Hypercube sampling strategy was employed due to the possibility that Crude Monte Carlo sampling would miss certain areas of the sample space. These missed areas could prove critical to the correct approximation of the search space. On the other hand, a Latin Hypercube sampling scheme (as discussed in Chapter 4) ensures that samples are distributed more evenly across the search space. A Sobol sequence would also be appropriate for this type of sampling but was not investigated for this work. A further refinement was also made to the Latin Hypercube sampling to include extra localised samples. When a functional (in terms of state machine representation) sample is found, another Latin Hypercube is generated around the sample point. A deviation of 5 % of the functional sample point was used for the boundaries of the hypercube with 20 extra localised samples drawn. This ensures that each functional sample point has neighbouring samples which would indicate the function dynamics in that area. The more information the networks receive regarding the function dynamics, the better the network should be able to interpolate. This resulted in 7750 samples being drawn for the AND gate, including non-functional samples, and 16330 samples for the OR gate. Notablye all the input and output values were scaled between 0 and 1 before being applied to the ANN.

The Fast Artificial Neural Network(FANN) library, used to implement the artificial neural networks, allows a wide variety of options. These include activation functions, type of training algorithm, steepness of activation functions and learning momentum. For the activation function various Sigmoid-like functions are available for which a steepness can be set using the appropriate parameter. Generally it was found that the Elliot activation function provided slightly better interpolation than the Sigmoid function when used with yield approximation. Mathematically, the Elliot function can be described as presented in equation 4.3.1.

$$y = \frac{x/2}{1 + |x|} + 0.5 \tag{4.3.1}$$

Furthermore, in terms of the training strategy it was found that batch training was not very effective when training the network for yield approximation. Incremental training was therefore used for all investigations. The FANN library also provided two more advanced training methods called RPROP and QUICKPROP. These methods were not investigated in this work. The parameters for the artificial neural networks are shown in Table 4.2.

Generally, the largest influence on the yield approximation was found to be the structure of the networks. The investigation therefore focused on the number of hidden layers as well as the number of processing elements in each layer.

Parameter	Value
Training Algorithm	Incremental
Activation Function	Elliot
Activation Steepness	0.5

Table 4.2: Selected FANN parameters for yield modelling.

The application of the full set of training data is called an epoch in ANN literature. It is difficult to know how many epochs one should train the network before using its approximation abilities. One option is to allow the network to keep on training until the mean squared error of the outputs compared to the training data stops reducing. This is not always the best strategy as the network might become "over-trained". This indicates that the network becomes so adept at approximating the training data that it loses the ability to interpolate new data (generalise). One way to mitigate this, as mentioned in [33], is to create another small set of test data that was generated independently from the training data. After each training epoch, the mean squared error of the test data is calculated to ascertain whether or not the network is any better at approximating untrained data. If the mean squared error increases after a training epoch, one can assume that the network is starting to over-train and is losing its ability to interpolate.

Bearing in mind the effect of over-training, an investigation methodology was created as shown in Figure 4.10.

Along with the training data, test samples were created independently using the same Latin Hypercube method. For the AND gate this consisted of 1780 samples and for the OR gate 6220 samples. Throughout the training phase, the current lowest mean squared error of both the training data and the test data were saved. The aim of this method is to find both network weights where the mean squared error of the test data and that of the training data are at a minimum. The minimum is selected if the lowest error value remains constant for the 1500 epochs.

The algorithm starts by creating a new network with the parameters shown in Table 4.2. The initial mean squared errors for the training data and the test data are then calculated and saved as the new lowest values. The counters are also reset to zero. A new training epoch now commences. If the lowest error of the test data has not yet been found, the test data is applied to the current network in a feed-forward manner (meaning that the weights are not updated and only outputs are generated) and the outputs are compared to the known outputs of the test data so as to calculate the mean squared error. If this new error value is lower than the previously stored value, the new value is stored as a new reference. The network is also saved in case the lowest error value had been reached. If, on the other hand, the new error value is higher than the saved error value, the counter is incremented. If this counter reaches the maximum number (in this case 1500), a flag is set to indicate that the test

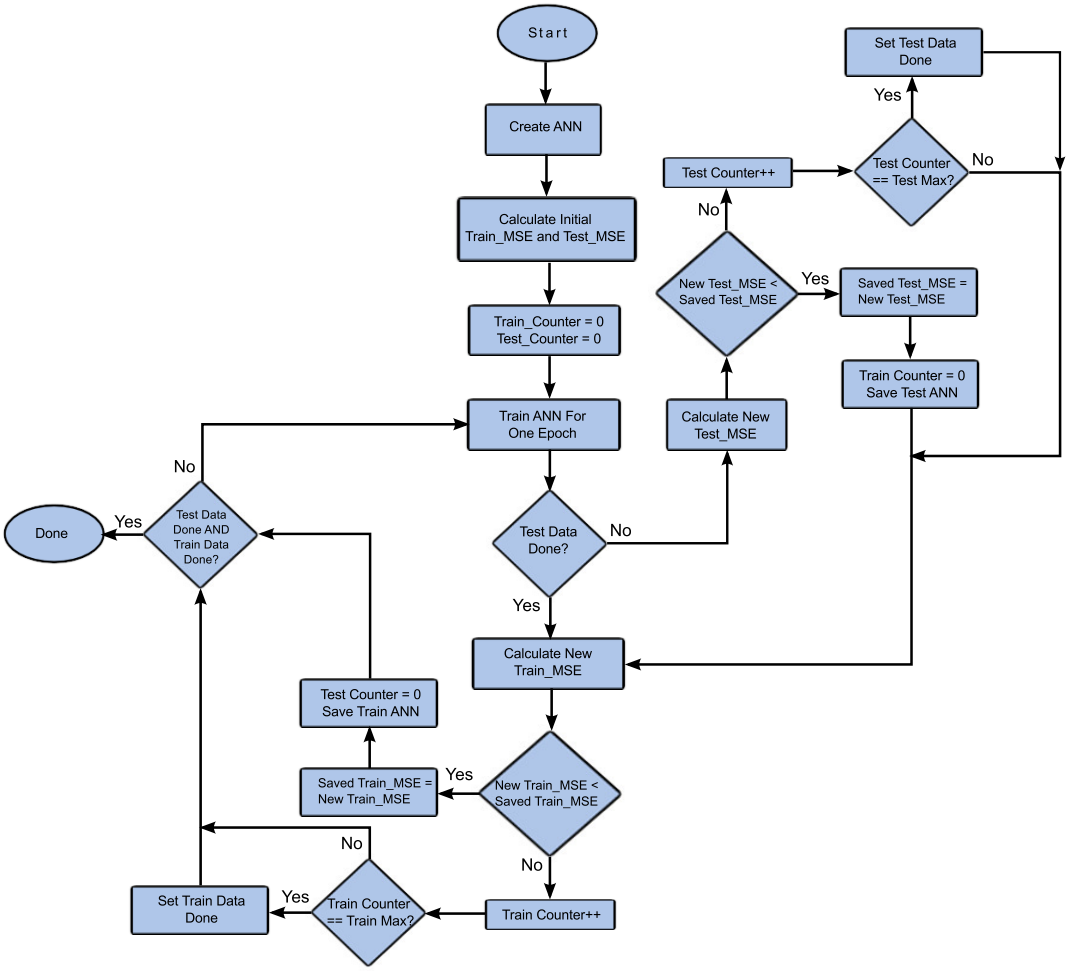


Figure 4.10: Training strategy flow diagram for artificial neural networks.

data phase has been completed and that the network is possibly over-trained. The last saved set of network weights will then have produced the lowest mean squared error for the test data.

The same method is applied to the training data in parallel. The new mean squared error is calculated and compared to the previous best mean squared error of the training data. If a better network is found, the saved error is updated and the network is saved for future use. If, however, no better network is found, the training counter is incremented and after 1500 counts have been reached, a flag is set to indicate that the training data portion have reached its lowest error value.

The aim is therefore to save the network weights at the state where the mean squared error of the test data is at its lowest and also where the training method cannot reduce the mean squared error of the training data any further. These two networks can then be compared by visualising circuit parameter sweeps as explained below.

It is noteworthy that the training algorithm itself is a randomised optimi-

sation algorithm since the initial weights of the system are randomly selected. It may therefore be necessary to run the network more than once to obtain useful values. Every network structure investigated in this study was therefore run five times. The hidden layers of the investigated structures for the two circuits are listed in Table 4.3. The input layers always constitute the number of inputs and the output layer is equal to one due to the fact that only one variable is approximated.

Network 1	(10)	Network 2	(20)
Network 3	(40)	Network 4	(10,10)
Network 5	(20,20)	Network 6	(40,40)
Network 7	(10,10,10)	Network 8	(20,20,20)
Network 9	(40,40,40)		

Table 4.3: Topologies of the investigated artificial neural networks.

Notably all the networks are fully connected, meaning that each input is applied to each processing element and each processing element is applied to every neighbouring processing element in the next hidden layer. A summary of the results are presented below, with the complete set of results available in Appendix B.

In Table 4.4, the results of the AND gate runs are presented. The lowest mean squared error for each structure in terms of the training data and the independent test data is presented. The run that was responsible for each of these values is also presented.

Number	Structure	Lowest Train MSE	Run	Lowest Test MSE	Run
1	(10)	0.0039	Run 1	0.0113	Run 3
2	(20)	0.0023	Run 1	0.0090	Run 1
3	(40)	0.0020	Run 1,2,5	0.0087	Run 5
4	(10,10)	0.0033	Run 3	0.0085	Run 3
5	(20,20)	0.0017	Run 2	0.0089	Run 4
6	(40,40)	0.0011	Run 1,3	0.0097	Run 3
7	(10,10,10)	0.0028	Run 1	0.0121	Run 4
8	(20,20,20)	0.0013	Run 1	0.0060	Run 3
9	(40,40,40)	0.00057	Run 4	0.0090	Run 4

Table 4.4: ANN yield modelling results for the RSFQ AND gate.

The value of the mean squared error of the training data varies substantially depending on which structure is used to approximate the yield. It is evident that any of the structures that only make use of ten processing elements in a hidden layer generally perform worse than their larger counterparts. In general, it would seem that the mean squared error of the training data is reduced when

more processing elements per hidden layer and more hidden layers are present. It is very interesting to note, though, that this trend is not visible in the mean squared error of the training data. In fact, both network numbers 6 and 9 seemed to perform worse in comparison with their 20 processing element counter parts. This might be due to the optimisation of the weights being more difficult in the presence of more free variables which could in turn have necessitated more runs to find an optimal weight distribution.

It is challenging to intuitivly know how these mean squared error numbers correlate with the ability to approximate the yield. In order to gain insight into the meaning of the errors, a visual inspection was made regarding the yield value over two free variables. This inspection entails the selection of two circuit variables after which a yield map is created by varying these variables over a specified range and calculating the yield using the Sobol sequence method desribed in Chapter 3 as well the artificial neural network approximation. B_2 and L_2 were chosen as free variables due to these components forming part of the fluxon storage loop which is crucial to the operation of the circuit. The range of each of the component values were divided into 25 equally spaced points and each combination of values were fed into the network while retaining the other components at nominal values. For comparison purposes, the two variables were first investigated using a Sobol sequence with 500 steps per point. A three-dimensional yield map was therefore generated. The result of the Sobol sequence sampling is depicted in Figure 4.11.

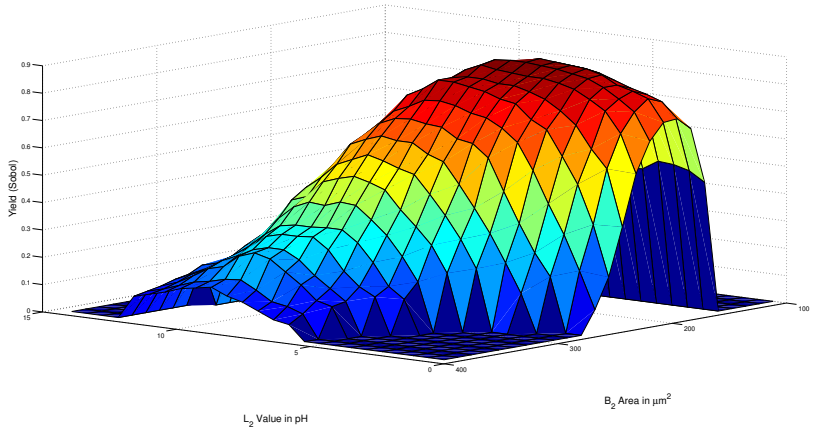


Figure 4.11: Sobol sequence yield map of junction B_2 and inductor L_2 for the RSFQ AND gate.

All the networks were compared to the yield map in Figure 4.11. The best comparison (subjectively chosen) for each hidden layer step is shown below. For example, the best comparison of all the runs of networks 1, 2 and 3 are indicated below. A single representative was therefore also chosen for networks 4, 5 and 6 and networks 7, 8 and 9.

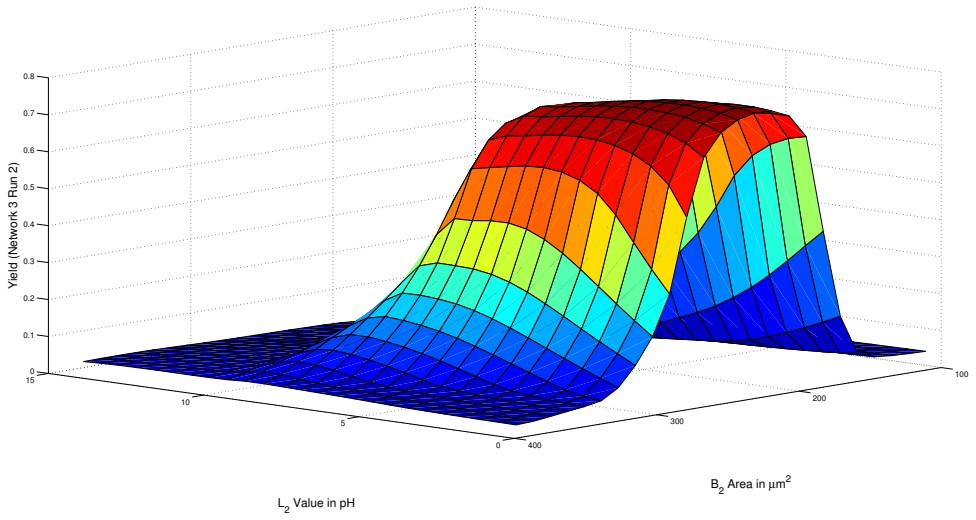


Figure 4.12: ANN yield map of junction B_2 and inductor L_2 using network 3 run 2 for the RSFQ AND gate.

The training error for Network 3 Run 2 was 0.0020 with a test data set error of 0.0091.

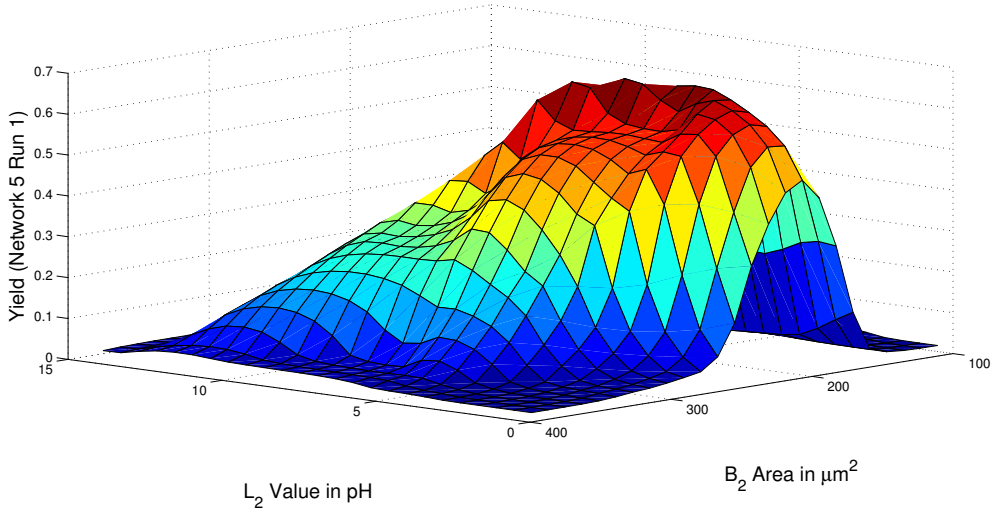


Figure 4.13: ANN Yield map of junction B_2 and inductor L_2 using network 5 run 1 for the RSFQ AND gate.

The training error for Network 5 Run 1 was 0.0020 with a test data set error of 0.0097.

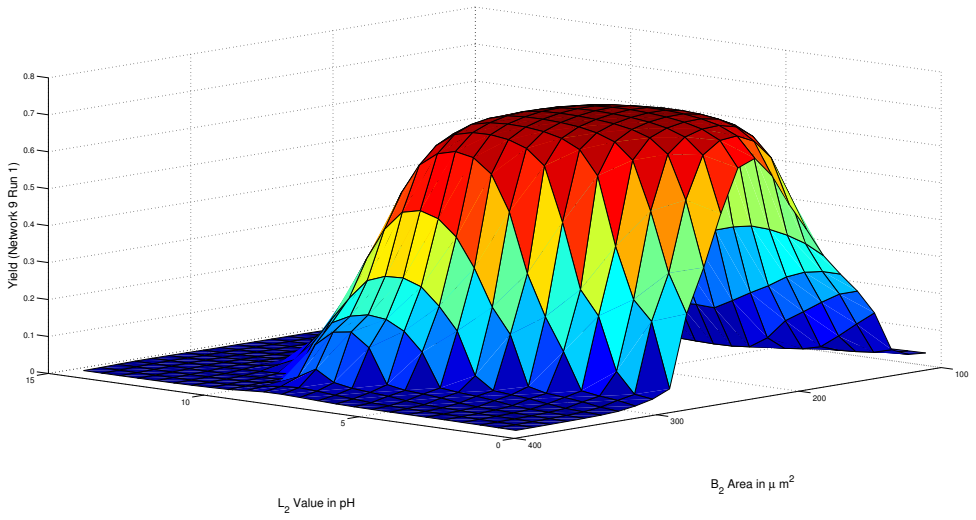


Figure 4.14: ANN yield map of junction B_2 and inductor L_2 using network 9 run 1 for the RSFQ AND gate.

The training error for Network 9 Run 1 was 0.00075 with a test data set error of 0.0103.

From a visual inspection of the yield spaces in comparison to the Sobol equivalent, it can be confirmed that a low number of processing elements generally does not provide adequate results. Furthermore, a single hidden layer tends to give a general, very broad approximation of the yield space, whereas more hidden layers are likely to add further detail to the approximation. There seems to be very little correlation between the visually pleasing approximations and these approximations that performed better numerically.

The same testing methodology was applied to the RSFQ OR gate and the results are indicated in Table 4.5.

Number	Structure	Lowest Train MSE	Run	Lowest Test MSE	Run
1	(10)	0.0208	Run 2	0.0310	Run 2
1	(20)	0.0176	Run 5	0.0272	Run 2
1	(40)	0.0158	Run 3	0.0278	Run 4
1	(10,10)	0.0176	Run 2	0.0283	Run 2
1	(20,20)	0.0115	Run 1,3	0.0274	Run 3
1	(40,40)	0.0065	Run 1	0.0258	Run 5
1	(10,10,10)	0.0157	Run 3	0.0259	Run 3
1	(20,20,20)	0.0109	Run 5	0.0283	Run 4
1	(40,40,40)	0.0043	Run 5	0.0276	Run 4

Table 4.5: ANN yield modelling results for the RSFQ OR gate.

The error values of the OR gate are generally higher than that of the AND gate. This could be due to the larger number of samples that were available for both the training data and the test data. Once again the data confirm that a small number of processing elements tend to render sub-standard results, which also tends to be the case for both the training and the test data. It is interesting too that the change from 20 to 40 processing elements in general produced a significant drop in the mean squared error of the training data. This could indicate that the OR gate, more so than the AND gate, needs more processing elements to correctly model its yield behaviour. This could further indicate that there is no real correlation between the number of processing elements and the number of input parameter values since the AND gate, with more input parameters, was sufficiently approximated with 20 processing elements.

A visual inspection of the yield map of B_7 and L_6 was also undertaken in relation to the Sobol sequence method presented in Chapter 3. These elements were once again selected due to their fluxon storage behaviour. A representative yield map is illustrated below for each hidden layer size, in the same manner as seen for the AND gate. For comparison purposes, a Sobol sequence method was used to generate the map shown in Figure 4.15 with each point consisting of 500 Sobol samples.

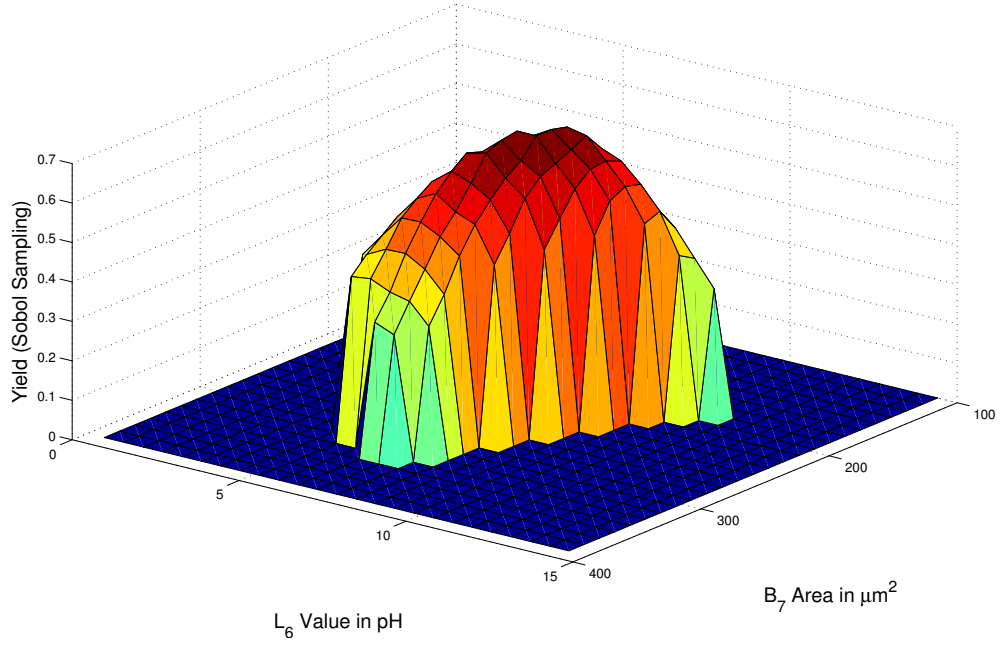


Figure 4.15: Sobol sequence yield map of junction B_7 and inductor L_6 for the RSFQ OR gate.

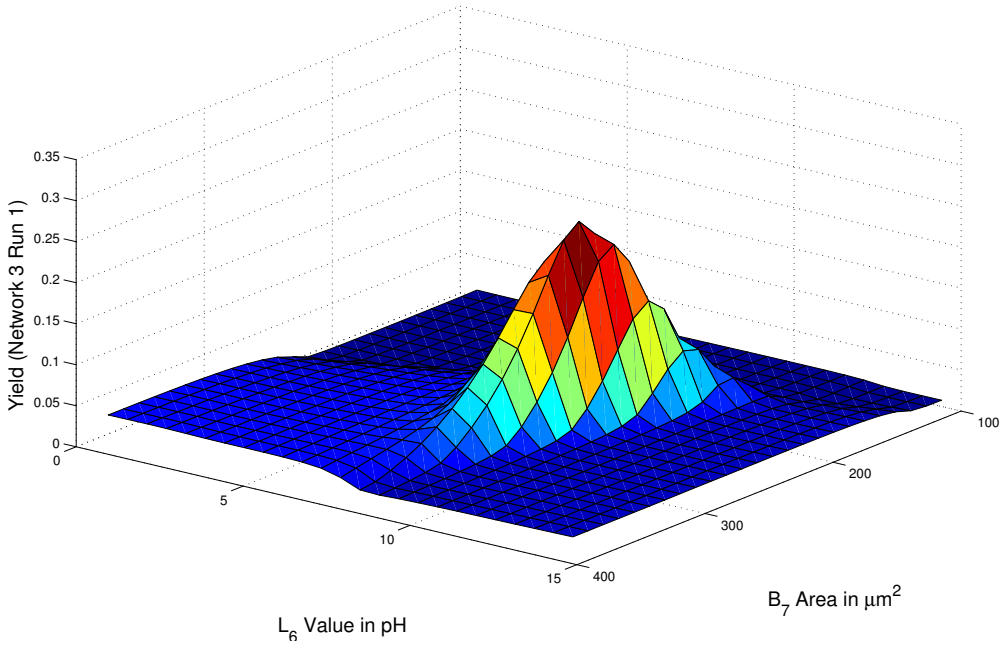


Figure 4.16: ANN yield map of junction B_7 and inductor L_6 for network 3 run 1 for the RSFQ OR gate.

The training error for Network 3 Run 1 was 0.0163 with a test data set error of 0.0279.

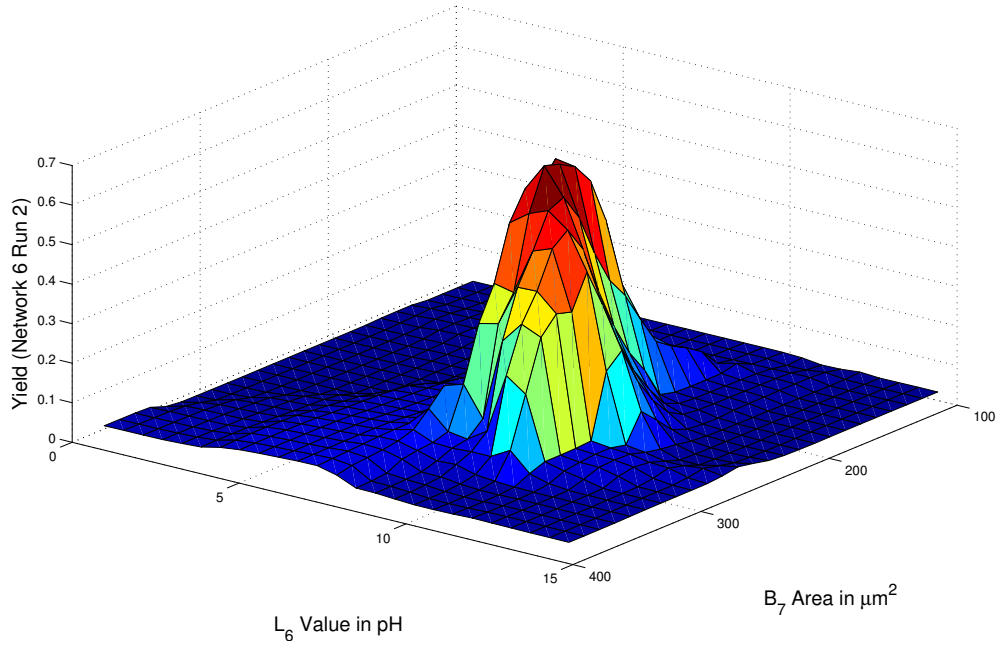


Figure 4.17: ANN yield map of junction B_2 and inductor L_2 using network 6 run 2 for the RSFQ OR gate.

The training error of Network 6 Run 2 was 0.0072 with a test data set error of 0.0277.

The training error for Network 9 Run 4 was 0.0049 with a test data set error of 0.0267.

Once again it is evident that single hidden layer networks can only approximate the general outline of the sample space. Also, visually there seems to be only a slight difference between the two and three hidden layer networks. In general, though, the three hidden layer network does seem to attenuate the yield values more compared to the Sobol sampling.

4.3.1 Conclusion

From the investigation discussed in this chapter it can be ascertained that the choice in network structure has a very significant influence on the performance of the artificial neural network. What is also evident, which can be surmised by investigating the full set of results in Appendix B, is that the back-propagation algorithm can result in diverse network behaviours for multiple training runs of the same network structure. It can therefore be surmised that the search space traversed by the back-propagation algorithm when training for yield might also be multi-modal (have more than one optimum). It is therefore crucial that networks are run multiple times to find the optimal solution. Alternatively, meta-heuristic optimisation algorithms could be investigated as a replacement

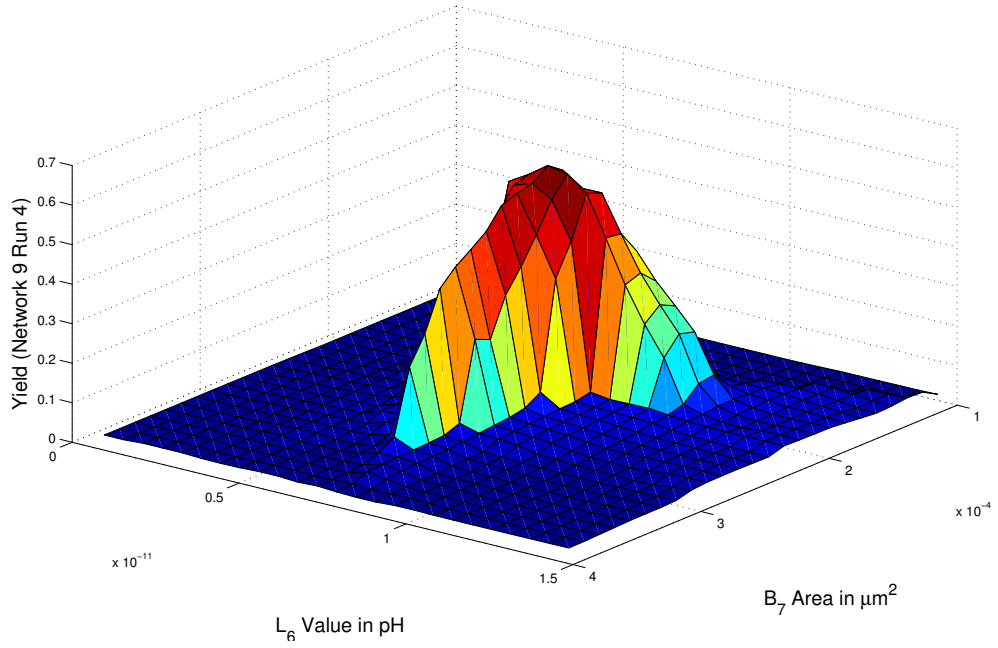


Figure 4.18: ANN yield map of junction B_2 and inductor L_2 using network 9 run 4 for the RSFQ OR gate.

for back-propagation because of their ability to find optimum points in multi-modal search spaces.

The usage of test data to ascertain the success of approximation tends to be inconclusive. The results do, however, show a slight indication that the networks which performed well in terms of the mean squared error and visual comparison to the Sobol sequence method do tend to have a lower test mean squared error, but no obvious trend is visible.

It was useful to visually inspect the network behaviour so as to ascertain whether the general yield space shape of the network output is close to what could be expected from a Sobol sequence Monte Carlo equivalent. Naturally this method is not conclusive, since not every part of the yield space can be investigated in this manner.

Given the results, the following guidelines should be followed when aiming to approximate the circuit yield of an RSFQ circuit with an artificial neural network:

- Gather training data that is well spread out across the search space;
- Gather a smaller independent set of test data;
- Start with a network structure with two hidden layers and around 20 processing elements per hidden layer. If the results are not favourable, increase the processing elements first before adding another hidden layer;

- Inspect the mean squared error of the test data set as well as the visual yield maps for signs of over-training; and
- Run the back-propagation algorithm multiple times.

It can be surmised that artificial neural networks have the ability to accurately model the circuit yield of an RSFQ circuit. The procedure to obtain a satisfactory approximation, though, is not very well understood and still relies on trial and error. The amount of training data required also tend to make the use of the network as a replacement for the Monte Carlo methods (such as discussed in Chapter 3) unlikely. This might be alleviated by using a different sampling strategy or a different network structure with a different interconnection matrix. More research is therefore necessary to ascertain whether artificial neural networks can be used to replace Monte Carlo methods in an efficient manner.

The networks do tend to render a fairly accurate overall representation of the yield space. The network can therefore be used as an investigative tool for algorithms that employ a large number of yield samples, such as optimisation algorithms. The speed at which yield samples can be generated using the trained network allows large amounts of statistical data to be gathered in order to quantify the performance of these types of algorithms.

This is the approach that is used to investigate various meta-heuristics in Chapter 6.

Chapter 5

State Machine and Timing Extraction

5.1 Introduction

The main aim of design centering is to ensure that a circuit is functional given all the possible manufacturing process variations. However, the meaning of "functional" needs to be explored before an attempt can be made to solve this problem. For digital logic circuits, such as those under investigation in this work, it is only natural that the implemented logical function should also be the primary functional attribute. For example, if the designer aims to implement a logical AND gate, the circuit response needs to be tested against the known logical functionality of an AND gate. Apart from the logical function implemented by the circuit, a designer might also define a functional bound on other parameters. This could include circuit timing (which is investigated in detail later in this chapter) and leakage current (which falls outside the scope of this work). Timing can influence the functionality of the circuit itself as well as adjacent logic circuits in a system composed of multiple logical cells. In pure asynchronous designs, such as [35], timing is a crucial functional specification for adjacent cells to function properly. Leakage current can also have a direct influence on biasing of adjacent circuits which could cause further circuit failure due to process variations.

In this chapter, a method consisting of various algorithms is presented that characterises RSFQ digital logic circuits in terms of logical functionality and timing. These characteristics can be used for the functional testing of circuits and for high-level circuit modelling.

Fundamentally, the logical operation of digital circuits can be represented by their respective state machine descriptions. This representation can either be based on a Moore machine or a Mealy machine. In a Moore machine, the output of the circuit is dependent only on the current state of the circuit whereas in a Mealy machine, the output is dependent on the current state

and the current input values of the circuit. For RSFQ circuits, with their picosecond pulse propagation, a Mealy representation tends to be more natural. In order to robustly test the logical functionality of any digital logic circuit, the designer needs to test each of the states in the state machine representation of the circuit for each of the possible circuit inputs. This is analogous with current FPGA and ASIC designs where circuit functionality is proven, given the complete circuit coverage from an applied test bench. In effect, every functional part of the circuit is exercised and tested against expected values.

One of the most prevalent methods of currently testing RSFQ circuit functionality is by manually creating a test bench in a SPICE simulator that ensures circuit coverage. Outputs of such a test bench are observed manually, or by means of computer script, in order to ascertain whether the circuit is functionally correct. Creating test benches manually can become a very time-consuming portion of circuit design for large circuits. This can be alleviated by using the method of state machine and timing extraction proposed in this chapter.The results of the extraction can then be examined for correctness by the designer or tested against other extracted state machines. The possibility of automatically generating state machine representations was suggested by [36], but to the best knowledge of the author, no known publication detailing this method has yet emerged.

Large parts of digital logic design in the semiconductor industry make use of hardware description languages to create complex designs. These languages, for example VHDL and Verilog, allow the designer to work on a higher level of abstraction in order to create designs that would be cumbersome to implement in instances where the full electrical representation of the circuit needs to be considered. In FPGA or ASIC designs, a designer usually creates system components by using register transfer level (RTL) or component instantiation in their selected hardware description language (HDL), before testing the logical functionality of their design by means of behavioural simulation. This is a simulation that does not take any circuit latencies into account and usually involves the designer creating a test bench of desired stimuli that will test the full range of circuit functionality. After the functionality has been confirmed, a timing analysis can be performed which can be static or dynamic. In a static timing analysis, the propagation paths for the clock, data and asynchronous signals are calculated throughout the design. The synchronous signals are checked against known timing requirements for the various sequential elements, such as flip-flops. Asynchronous signals, on the other hand, are tested against recover-removal requirements. It is important to note that the asynchronous signals generally refer more to asynchronous set-resets of sequential circuit elements or to input signals from an unknown clock domain, rather than a fully asynchronous design. A static timing analysis therefore does not consider the logical functionality of the circuit, but rather only tests the sequential elements for timing violations. In contrast, a dynamic timing analysis depends on an input test bench created by the designer to logically

test the circuit by considering the appropriate component and path latencies. For FPGAs this is achieved by first synthesising the circuit to be tested which then translates the RTL and instantiation code to a net list of physically implementable constructs in the FPGA. The latencies through these constructs are known to the simulator which subsequently allows the simulation to test for timing violations. Further timing information can be obtained after the place-and-route cycle when the various components of the net list are placed in the FPGA fabric, and path latencies can be confirmed. Generally, a dynamic timing analysis is more accurate than a static timing analysis due the extra logical verification, but it does take longer to simulate and is also more cumbersome to implement since the designer first needs to create a test bench.

The same general design flow applies to RSFQ circuits and the semiconductor industry. Some studies have been undertaken by various researchers on the synthesis [37, 38] and place-and-route [39] stages of the RSFQ circuit design process. There are, however, some fundamental differences between RSFQ and semiconductor technology that should be considered. Clock timing in RSFQ circuits tends to be very challenging due to the picosecond pulses used for information passing and also the high operation frequencies [40, 41]. Due to this design challenge, it has not yet been conclusively established whether a synchronous or an asynchronous design methodology would be most beneficial for large scale RSFQ integration. It is therefore important that asynchronous methodologies, such as reviewed by [5], as well as synchronous methodologies should be possible with whatever high level design constructs are available.

Various researchers also proposed using high-level HDL modelling for RSFQ circuit design, such as in [36, 42, 43, 44, 45] [46]. The presented work, however, offers a method for the automatic generation of these models given the SPICE representation of a circuit and other user inputs. This model can be used for an asynchronous or a synchronous circuit design.

5.2 Method Overview

A method is presented in [47] to automatically extract the state machine representation of an RSFQ digital logic circuit, along with its timing characteristics in order to alleviate the cumbersome nature of test-bench generation for circuit functionality verifications and to generate HDL representations for a high level circuit design. As explained later in this section, the method revolves around identifying the inductive loops that store fluxons, and monitoring these loops as inputs are exhaustively applied in all circuit states. The method is implemented in C++ and forms part of the larger software library which is expanded upon in Appendix A.

A functional SPICE deck and its input and output nodes are supplied to the program at start-up. Currently, a practical circuit settling time should also be supplied. This is the time (which need not be accurate) that should

lapse after each input pulse application to allow all circuit transients to settle. In future versions of the method this requirement will most likely be omitted due to the ability of the program to identify the passing of transients in the circuit. The various other input options that are available to the designer are elaborated on in the detailed method discussion.

The input SPICE deck is first parsed and flattened (if applicable) after which an undirected graph is constructed for loop identification. Then the undirected graph is used to identify the shortest possible loop containing each circuit element. These loops are all monitored in the initial state machine extraction process before the method verifies which loops are in fact used for fluxon storage loops.

After the completion of the loop identification process, the extraction process can commence. An SFQ pulse is applied, one at a time, to each of the circuit inputs. After the circuit transients settle, the identified loops are monitored for fluxon storage or loss. A change in stored fluxons indicates that the circuit underwent a change in state. This new state (and the stimulus that led to the new state) is stored for later investigation. After a pulse is applied to each of the circuit inputs, the method selects one of the new states to investigate. The method then applies the stimulus that led to the state under investigation after which pulses are once again applied to the inputs individually. All transitions in the circuit states are therefore investigated.

Throughout the extraction process the output nodes are investigated for the generation of output pulses, which is also stored with the state information. Timing characteristics can also be extracted once the state machine representation is known. This is necessary in instances where dynamic HDL timing models need to be constructed for a high-level design. The timing characterisation builds on the definitions of critical and delay timings as proposed by [43]. Two different timing characterisation methods are available to the designer: a method that focusses on empirical timing information extracted by means of repeated SPICE runs, and another that makes use of identifying circuit transients. These timing methods are discussed in more detail later in this chapter.

The designer can elect to use the generated state machine representation for failure testing or for HDL generation. For failure testing, the state machine representation of derivative circuits (changed due to Monte Carlo variations or optimisation techniques) can be compared to the nominal state machine representation to prove logical equivalence. For HDL generation, the state machine representation and the timing characteristics can be employed by the method to automatically generate an HDL model.

The assumption is made that the correct functionality of the circuit does not depend upon the simultaneous arrival of SFQ pulses. This is, however, not a major limitation given the difficulty in aligning the picosecond wide SFQ pulses in practical circuits. If this functionality is required in the future, further expansions of the method would be necessary.

5.3 Flux Calculations

The core of the method is the quantisation of flux in RSFQ circuits. The only way for an RSFQ digital logic circuit to change its behaviour is by either storing or releasing a fluxon. This coincides with the state machine representation of these circuits. It follows that by investigating the inductive loops of a circuit for the storage or loss of fluxons due to the effects of the application of input pulses, these state machine representations can be constructed.

The first step entails the ability to calculate the flux change in inductive loops. The magnetic flux through a loop can be calculated using the current and loop inductance values as shown in equation 5.3.1:

$$\Phi = \sum_{n} I_n \times L_n \tag{5.3.1}$$

where n is the number of components in the loop.

Considering Figure 5.1, if one would like to identify the instantaneous amount of flux in loop B_1—L_1—B_2 then by using a SPICE simulation one could easily log the simulated current through the three respective elements in order to derive the value of L_1. The inductance of a Josephson junction, however, changes depending on the amount of current passing through the junction [9]. The inductance can be approximated as follows, depending on the instantaneous current:

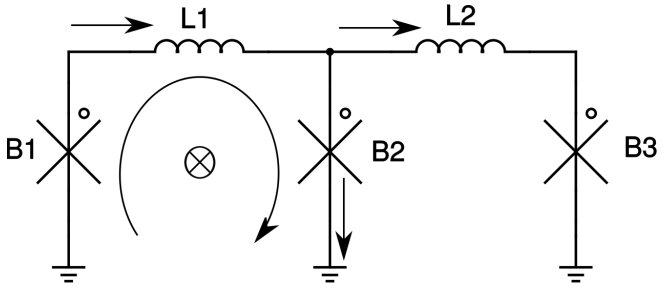


Figure 5.1: Calculation of change in magnetic flux.

$$L = L_1 \frac{\arcsin 2\pi \frac{i}{I_c}}{\frac{i}{I_c}} \tag{5.3.2}$$

where i is the instantaneous current and I_c is the critical current of the junction. Also

$$L_1 = \frac{\Phi_0}{2\pi I_c} \tag{5.3.3}$$

with $\Phi_0 = 2.0679e^{-15}Wb$.

The loops can be investigated for fluxon storage by calculating equation 5.3.1 after each input application, assuming that the previously mentioned loop is currently storing a fluxon. It is important to note that when investigating L_1 for flux storage, loop B_1—L_1—L_2—B_3 should have the same flux value as the mentioned storage loop. This is due to the quantisation of flux in superconducting circuits. The increase in current through B_2 due to the stored fluxon causes a net change in flux in loop B_2—L_2—B_3. If this flux does not equate to a single flux quantum, a shielding current will form in L_2—B_3 to expel the change in flux. This shielding flux will therefore be equal and opposite to the contribution of the flux caused by B_2 in loop B_2—L_2—B_3. For loop B_1—L_1—L_2—B_3 this will therefore add the same contribution as B_2. This is an important observation since one can therefore investigate any formed loop containing L_1 in order to ascertain whether this component is part of a fluxon storage loop. Practically, though, the smallest loop possible will be used for the investigation process in order to minimise calculation times.

An interesting exception to the above mentioned calculation methodology can be seen in Figure 5.2. Supposing that the loop inductance of B_1—L_1—L_3—B_2 is not sufficient to store a fluxon but loop B_1—L_1—L_2 is able to do so, then when a fluxon is stored in the said loop, a shielding current forms through the elements L_3—B_2 to counteract the net flux gain in loop L_2—L_3—B_2 as previously explained. An exception occurs when the amount of shielding current required to counteract the effect of L_2 in the loop is greater than the critical current of B_2. Through simulation it can be verified that B_2 does not enter its resistive state, but rather that the excess current is shunted through L_4—B_3. This phenomenon does not cause a problem for the fluxon identification method, though, since the sum of the net flux gain in loops B_1—L_1—L_3 —B_2 and B_2—L_4—B_3 still equates to that of a single flux quantum. In effect, the method investigates adjacent loops if a flux change is identified that is either a single flux quantum or no flux change after an input application.

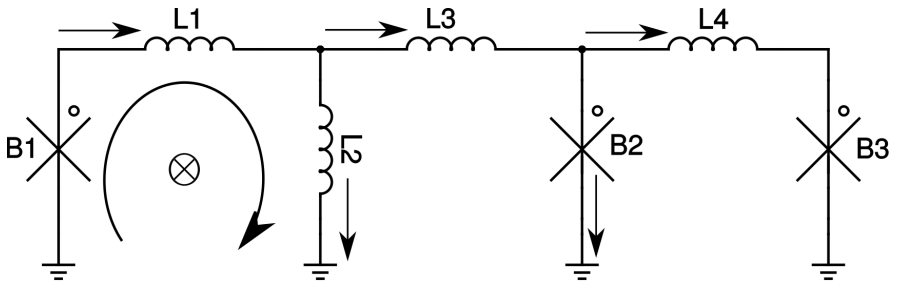


Figure 5.2: Example of a flux calculation exception.

5.4 Detailed Method Description

This section describes the details of the extraction method.

5.4.1 User Inputs

The following user inputs should be supplied for the correct extraction of the state machine representation:

- A functioning SPICE deck containing the circuit under investigation in a format parsable by JSIM [48] or JSPICE. Other simulation formats can relatively easily be incorporated into the software library.
- The input nodes of the circuit. These are the nodes that will be used for the application of input pulses.
- A Josephson junction in the input path of each input node from which circuit latency is calculated. Generally a junction just before the supplied input node is used for this purpose to ensure that the true circuit latency will always be slightly less than the calculated latency. This junction would normally be situated in an adjacent logic cell or in a DC-SFQ circuit used to generate stimuli. In future implementations of the method, this requirement can be reinvestigated. It should be possible to use the created undirected graph to automatically identify the first parallel junction before the supplied input node.
- The output junctions that should be monitored for output generation and for delay (latency) characterisation. In future implementations of the method an output node can first be supplied and then a parallel junction can automatically be identified for the monitoring process.
- The time after each input that is required for all the circuit transients to pass. This requirement can also be removed in future versions of the implementations by initially monitoring the current passing through the last element in the longest possible path in the generated undirected graph.
- Any supplied user rules (optional). To understand the need for user rules consider Figure 5.3.

In general a circuit is designed so that any sequence of input pulses can be applied without influencing the logical functionality of the circuit, regardless of the circuit state. For example, in Figure 5.3 the series junction B_2 is used to "throw-out" input pulses if a fluxon is already stored in loop B_3—L_3—B_4. The designer can, however, elect not to include B_2 if a decision is reached that

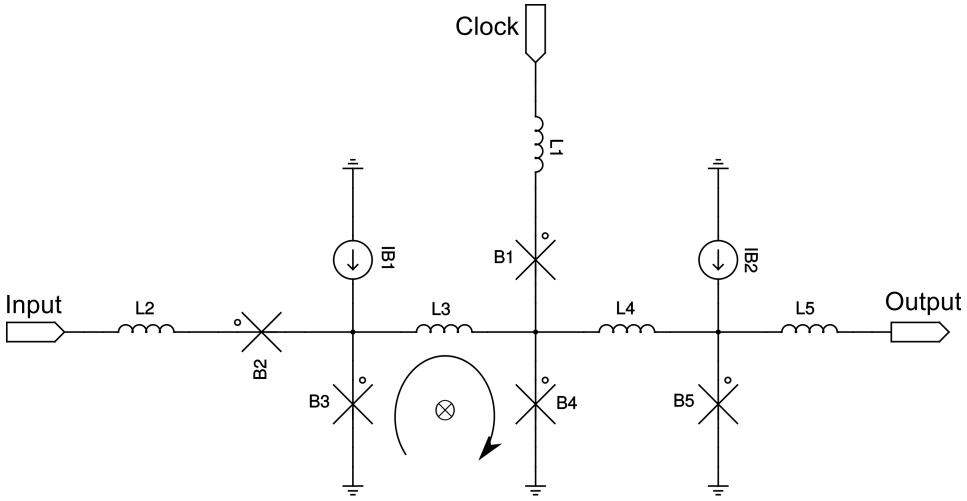


Figure 5.3: Circuit schematic of an RSFQ D-Flip-Flop.

the circuit is unable to receive two subsequent input pulses without a clock pulse arriving first.

In order to facilitate this requirement, the user can supply rules to the extraction method. For instance, a rule for Figure 5.3 could be that a clock pulse has to arrive after an input pulse before another input pulse can be applied. Practically this would result in a maximum rule which should be supplied to the method as follows:

$$rule\ max\ in\_node\ 1\ reset\ clk\_node \tag{5.4.1}$$

where in_node would be the input node number and clk_node the clock node number. The number after the node to which the maximum rule applies designates the number of pulses allowed before the rule is violated. The reset keyword designates that the following node resets the maximum rule.

In general the following keywords are available:

- $max\ A\ B$: maximum number of sequential pulses on node A is B.
- $after\ A$: the previously supplied maximum rule only applies after a pulse on A.
- $reset\ A$: the previously supplied maximum rule should be reset after a pulse on A.
- $override\ A$: the previously supplied maximum rule is ignored if a pulse on node A arrives after a pulse supplied with the $after$ keyword.

In future implementations of the method, functional differences could be supplied with the input node numbers. Input nodes can, for example, be classified as clock inputs for synchronous circuits which can automatically be used to reset input pulse rules.

5.4.2 SPICE parsing, flattening and graph creation

The next step in the method is to read and parse the SPICE deck. Any hierarchical instantiations are removed by creating a unique component (with unique connected nodes) for each component within a sub-circuit instance. This will ensure that loops that span multiple instanced cells are correctly identified. The designer is also able to specify sub-circuit instances to ignore for the flattening and extraction process. This is useful for peripheral circuits that are used to excite the circuit under test but does not form part of the functionality of the said circuit. Circuits such as DC-SFQ cells and JTLs are usually ignored for the extraction process as they are generally used to generate stimuli and do not form part of the tested circuit functionality. An undirected graph is created from all the flattened components, while ignoring any components in sub-circuit instances that appear in the ignore list. Any component that only has one connected node due to the peripheral cells being ignored (such as input and output inductors) is excluded from the graph. The nodes of the undirected graph correspond to the nodes of the SPICE representation while the vertices of the graph correspond to the two terminal components (inductors or Josephson junctions).

An undirected graph of the D-Flip-Flop presented in Figure 5.3 is depicted in Figure 5.4.

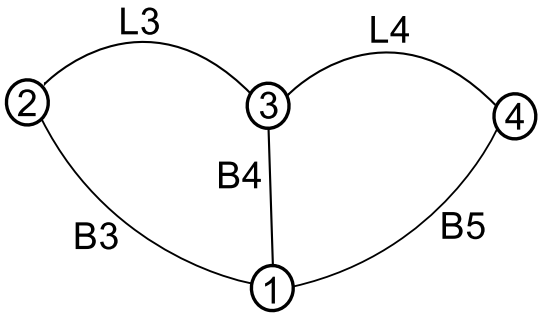
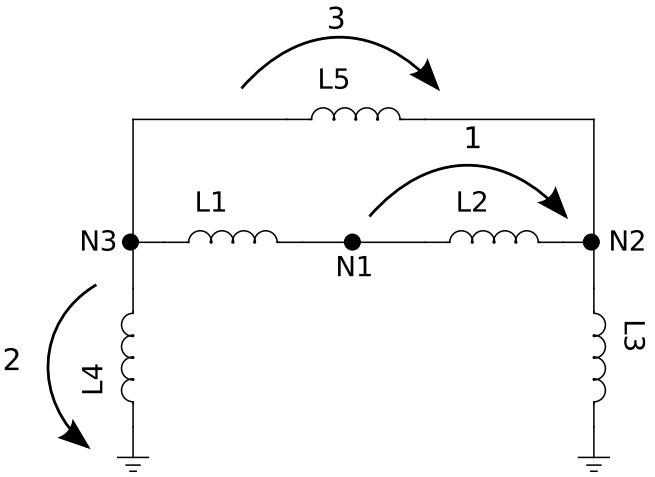


Figure 5.4: Un-directed graph of D-Flip-Flop components.

5.4.3 Inductive Loop Identification

The aim of the inductive loop identification is to find the smallest loop for each inductor in the circuit under test. This is achieved by first selecting an un-investigated inductor and arbitrarily assigning one of its nodes as a target node and the other as a source node. The investigation takes place in parallel from both of these nodes, in a direction away from the component itself. In each iteration of the algorithm each path (target and source) adds a single unselected vertex and node pair to its path list. If more than one vertex is selectable (for example a node connecting three components), a new path is

created by copying the path that caused the current node to be reached and adding the remaining unselected vertex. In the next iteration of the algorithm a new vertex is selected for the new path as well as for the initial paths. After each iteration, the identified nodes are compared to ascertain whether two of the paths have reached a common node. The first pair of paths that reach a common node is necessarily the shortest path (though other paths might exist that contain the same number of vertices). This is due to the fact that each path only adds one vertex-node pair during each iteration, therefore keeping all the paths the same length. It is important to note that once a loop has been identified, none of the loop components need to be investigated again since the identified loop for each of these components will contain at least the same number of elements as the initial loop.



Iter	Path 1	Path 2	Path 3
1	N1	N3	N3
2	N2	GND	N2

Figure 5.5: Finding the smallest loop containing inductor L_1.

For example, when one considers Figure 5.5 where the aim is to ascertain the smallest inductive loop containing $L1$, the target and source node can be selected as $N1$ and $N3$ respectively. At the first iteration, three paths are created; one path starting from $N1$ and two paths from $N3$ since the last named node has two unselected vertices. In the next iteration each path adds one component away from the component under investigation. Path 1 adds $L2$ and finds node $N2$ while path 2 adds component $L4$ and finds the ground node. Path 3 adds component $L5$ as its vertex and also finds node $N2$. After each iteration the identified nodes are investigated in a search for commonality. In this case both path 1 and path 3 found node $N2$. The components determined by path 1 and path 3 can now be added together to form the shortest inductive loop containing $L1$, that is, $L1$—$L2$—$L5$.

5.4.4 State Machine Extraction

After the loop identification process, the state machine extraction process can commence. Initially, it is not known which of the identified loops are used as storage loops by the circuit. Therefore, all the components that form part of one of these identified loops are investigated through the extraction process. For the purpose of failure testing, any subsequent extraction process can only investigate the loop components that are known to form part of a storage loop. This provides a necessary processing performance increase when performing taxing investigations, such as Monte Carlo runs.

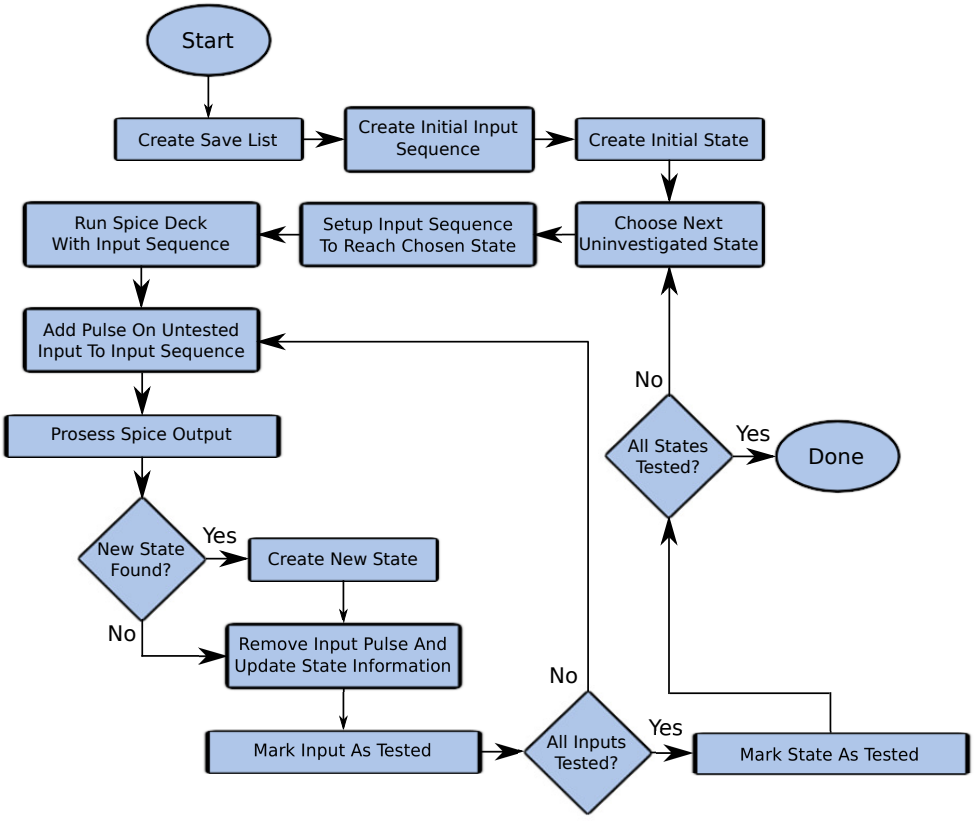


Figure 5.6: State machine extraction flow diagram.

The extraction process is described in conjunction with the flow-diagram in Figure 5.6, as presented in [47].

The first step of the algorithm is to create a list of components to be investigated in the extraction process. This list includes all the components which form part of the identified inductive loops and the input and output junctions supplied by the designer. The transient current values of the loop components are stored for flux calculations while the input and output junction flux values are stored in order to identify output pulse generation and to calculate the latency from the input to the output pulses. An initial input sequence and

initial state are then created. The input sequence contains the sequence of pulses that caused the circuit to reach the state under investigation as well as any new stimuli that should be applied to the state. For example, an input sequence could have been a pulse on $InputA$ at time X followed by a pulse on $InputB$ at time Y, and so on. In examining the flow-diagram, it is evident that after a state investigation phase has been completed a newly identified state that has not yet been investigated is selected as the test state. The input sequence is then populated with the nominal sequence of input pulses that caused the state under test to be reached. At start-up, only the initial state is available for testing which by definition requires no input pulses to be reached. Hence the input sequence of the initial state would be empty.

Nominal flux values are required for each state in order to calculate flux changes caused by input pulses. Since the initial state was not identified by the algorithm through the normal input application process as explained below, an initial SPICE run was used to ascertain what the nominal flux values of this state are.

The process then continues by selecting an untested input to the circuit. Thereafter, a pulse is added to the input sequence and applied a certain amount of time after the previous pulse application in the sequence so as to ensure that all circuit transients have dissipated. This amount of time is the settling time provided by the user during the set-up procedure. A SPICE deck is then automatically generated with the applied input sequence consisting of the current and phase values of the previously created save list which are identified as variables to log.

Once the SPICE run is completed the new flux values are calculated by using the last stored value of each current measurement. Here the assumption is that the settling time supplied by the user was sufficient to dissipate any current transients in the circuit. In future versions of the implementation a test could be added to calculate the finite difference of the last stored value in a measurement. If this finite difference is above a certain value (indicating that transients are still active), the designer can be notified or the SPICE deck can be rerun for a longer period of time. The calculation process is done using equation 5.3.1 to ascertain whether any fluxons have been stored or lost due to the application of the input pulse.

The input and output junction measurements are also processed for junction switching. The stored phase values are stepped through, and if 80% of a 2π phase shift is detected, a switch event is logged. The time of the input and output switched values are also logged. The difference between the output switching time and the input switching time is used as the latency for output pulse generation.

After the calculation phase has been completed all the previously identified states are searched so as to compare them with the changed fluxon values (if any). If a previous state is identified or if no fluxon change has occurred (indicating that no state change took place), the state-under-test transition

table is updated to log the transition. If no previous state is identified, a new state is created containing the fluxon identifiers that were determined during the simulation process. This state is then added to the un-investigated state list for later investigation.

After the state-under-test transition table has been updated, the last applied input is removed from the input sequence and marked as tested. Another untested input is then selected and added to the input sequence after which the SPICE deck is once again generated and the measured data are processed. After all the inputs of the state-under-test have been investigated, the state testing is flagged as completed. Next, an untested state is selected for the input application.

The assumption is that the initial state did not contain any stored fluxons. If a fluxon was being stored in this state at the time, a subsequent state would have been identified to gain a negative fluxon. After the extraction process has been completed, a normalisation process is performed to remove the negative fluxon from the relevant states and to add the same fluxon to all the other states.

When no further untested states are identified the extraction process is complete.

5.4.5 Critical Timing Extraction

In general, the critical timing of a circuit is defined in terms of the minimum (or maximum) time allowed between related input pulses. The relationship between these pulses indicates that both have an effect on the same functional elements in a circuit. In semiconductor devices the critical times are usually described in terms of set-up and hold times. However, due to the prevalence of synchronous circuits in semiconductor technology, these times are always regarded in relation to the clock signal. As depicted in Figure 5.7, the set-up time is the time before the clock edge (usually rising) when the data have to be stable.

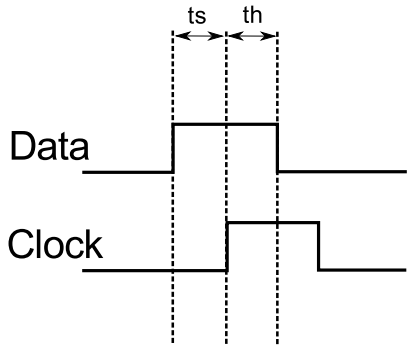


Figure 5.7: Illustration of set-up and hold times.

The hold time is the time after the clock edge when the data have to remain stable. In the case of flip-flops, a timing violation might either sample the data pulse incorrectly or cause the flip-flop to become metastable.

In terms of RSFQ circuits, critical times remain a relationship between input pulses. The main difference is that the critical times may differ substantially depending on which state the circuit is currently in. This effect is caused by the variability of inductance of a Josephson junction, depending on the bias current as shown in equation 5.3.2. A stored fluxon leads to a ripple effect, generating shielding currents to expel any non-quantised flux. These shielding currents can therefore have an effect on the bias currents of junctions that are not situated close to the fluxon storage loop. The change in junction inductances then have a further effect on the transient behaviour of the circuit due to equation 5.4.2.

$$v = L\frac{di}{dt} \tag{5.4.2}$$

It is therefore essential to investigate critical timings in terms of the state of the current circuit as proposed by [43].

Two approaches were investigated for the extraction of critical timings.

The first empirically tests the minimum delay between subsequent pulses by using SPICE runs to ascertain when the functionality of the circuit is affected. These tests are performed for each pulse pair and for each circuit state.

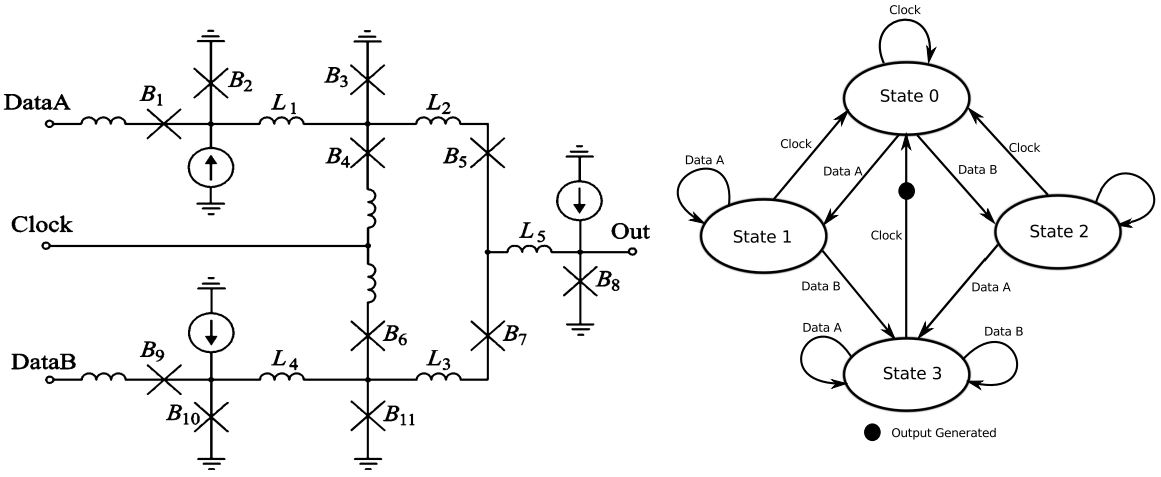


Figure 5.8: A simplified RSFQ AND gate with Mealy state diagram.

When considering the AND gate and its Mealy state machine representation in Figure 5.8, it is evident that all the shunt resistances and parasitic elements were removed for the sake of clarity. The four possible states are governed by the storage (or lack of storage) of fluxons in the two storage loops of the circuit, B_2—L_1—B_3 and B_{10}—L_4—B_{11}. For each of these states, the minimum allowable time between input pulse pairs needs to be calculated. This

can be achieved relatively easily since the correct behaviour is known due to the state machine extraction method described earlier. For example, assuming that the circuit is in State 0, then no fluxons are stored. If the relation between the *DataA*—*Clock* sequences is investigated, the stored state machine is aware that it should move to State 1 under functional circumstances and then return to State 0 without generating an output pulse. The first step in locating the minimum allowable delay is to ascertain whether a timing relationship exists at all. In order to test this, *Clock* is applied to the circuit for the shortest time possible (usually a picosecond) after *DataA*. Depending on the component values and manufacturing process, the currents in $B3$ and $B4$ might not have stabilised sufficiently by the time the *Clock* pulse reaches B_4, possibly throwing the *Clock* out of the circuit through B_4 switching. The state transitions identified by the state machine extraction in this scenario would be only a transition to State 1 which would not compare correctly to the nominal state machine representation. The algorithm can therefore assume that a relationship exists between the $DataA \rightarrow Clock$ sequence which should be investigated further. The investigation now returns the time delay between the input pulses to that supplied by the designer after which a binary search ensues. The time delay is therefore initially halved. If the circuit is still functional after half the delay, half the difference between the last functional time (in this case half the supplied time) and the last non-functional time (the 1 picosecond delay) is subtracted from the current delay. If not, half the difference between the last functional (the supplied time) and the last non-functional time (in this case half the supplied time) is added to the delay. The algorithm iteratively updates the functional and non-functional times by either adding or subtracting half this value from the current delay in order to create a window around the critical time. The designer is able to supply a acceptable uncertainty (windows size) after which the algorithm will return the last working value as the critical time relationship between the two pulses.

It is important to note that for these two inputs the sequence $Clock \rightarrow DataA$ also has to be investigated. In this sequence the nominal state machine remains in State 0 after the application of the *Clock* pulse and then transitions to $State1$ after the arrival of *DataA*. This is exactly what was determined from the 1 ps delay example which therefore indicates that no relationship exists between the input pair in this sequence.

The main assumption of this method is that all the critical timing information of the circuit can be described in terms of the relationship of input pairs only. Input triplets and n-tuples in general are not investigated at all. For example, it is assumed that when the sequence $DataA \rightarrow DataB \rightarrow Clock$ is applied to the circuit, the delay between $DataA \rightarrow DataB$ does not have an effect on the critical timing between $DataB \rightarrow Clock$. This is a very difficult assumption to prove. In effect, to ensure that this is not the case, every signal input in an n-tuple sequence needs to be investigated so as to create an exhaustive critical timing model which would incur a significant performance

penalty.

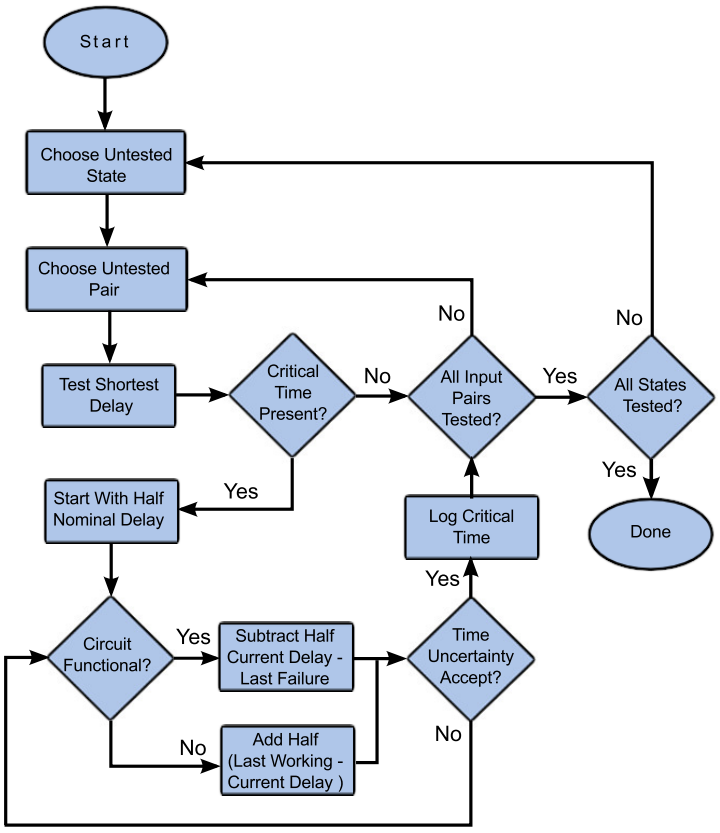


Figure 5.9: Empirical critical timing extraction algorithm.

An overview of the first method is presented in Figure 5.9. At start-up, an untested input pair of an untested state is selected for investigation. First a SPICE run is executed to test the shortest delay possible between the two input pulses (usually 1 ps). The resulting state can then be tested against the nominal extracted state machine representation for circuit functionality. If the run was found to be functional, no critical timing relationship is present and the next untested input pair is investigated. If the run was non-functional, a new delay is selected starting at half the nominal delay provided by the designer. A binary search algorithm is then applied as explained above. After each run the current critical time uncertainty is tested (window size). This is the value of the last function delay minus the last non-functional delay. If this difference is smaller than the acceptable uncertainty, the last working delay is accepted as the critical time for the input pair. Once all pairs in all states have been tested, the algorithm is complete.

The second method investigates the currents in the system to ascertain when all transients have passed after the application of an input pulse. Before applying such a method, the term "stability" needs to be defined for the

current application. Currently equation 5.4.3 is used to ascertain whether a current under investigation is stable. In effect, a finite difference is calculated by using the difference between subsequent current values (normalised to the first sample) and testing them against a threshold. At present no theoretical approach exists to identify the threshold value. This value depends on the SPICE simulation time-step, the circuit under investigation and the manufacturing process being modeled by the current simulation. For example, it was found that using the $1kA/cm^2$ process of the Institute of Photonic Technology (IPHT) [49], a value of 0.01 tended to render very repeatable results when using a SPICE time-step of 1 ps.

$$\frac{x(t-1)-x(t)}{x(t-1)} < threshold \tag{5.4.3}$$

One step of stability as described in equation 5.4.3 also does not ensure that the circuit is in fact stable. The algorithm has considered the possibility of slow varying signals that could still cause instability in the circuit operation. Equation 5.4.3 was therefore modified to function with a number of samples as presented in equation 5.4.4 in order to mitigate this situation.

$$\frac{x(t-1)-x(nt)}{x(t-1)} < threshold\ should\ be\ true\ all\ n\ where\ n = 1...k \tag{5.4.4}$$

For this equation, the initial sample is held as a reference and normalisation ratio but the inequality needs to be met for a number of subsequent samples before the algorithm acknowledges that stability has been reached. A value of 5 for n was found to be robust for the manufacturing process and SPICE time-step indicated previously.

The next variable to investigate when using the stability method is the decision about which components to test for current stability. The most robust solution is to test every component in the circuit. This, however, incurs a significant processing penalty on the algorithm and for many circuits this will result in large critical times. Considering the AND gate in Figure 5.8 once again, it is apparent that critical times, as stated before, are a relational attribute between input pulses. If, for example, one investigates the critical time between $DataA$ and $DataB$, one recognises that the circuit was designed so that these two pulses can arrive at the same time without affecting the functionality of the circuit. The stability algorithm testing all the components would, however, assume that the stability of all the components could possibly have an effect on $DataB$ after the arrival of $DataA$. The critical time would therefore be calculated as the time that the last component took to stabilise (possibly B_8 in the circuit). This would therefore result in unnecessary timing violations and a large performance penalty if these critical timings are to be taken into account for further circuit design.

On the other hand, a sub-set of components containing the storage loops might generate a more realistic critical time. The algorithm could be modified to only investigate the storage loops that change their storage characteristics (gain or lose a fluxon) due to the application of the second input pulse in the critical timing pair. This information is readily available due to the state machine extraction process described earlier. For example, consider the critical time between the input pair $DataA \rightarrow Clock$. An input pulse on $DataA$ causes a fluxon to be stored in its associated storage loop thereby transitioning the state machine from State 0 to State 1. A subsequent $Clock$ pulse should release the previously mentioned stored fluxon and return the state machine to State 0. Since the state machine fully describes the functionality of the circuit, and also since the state machine is implemented by means of the fluxon storage loops, one might argue that the critical timing should only account for settling times of relevant storage loop components. In the example the algorithm would therefore monitor components B_2, L_1 and B_3 for current stability before allowing the $Clock$ pulse to be safely applied to the circuit.

The flaw in this argument, though, is that the effects of series junctions are ignored. If the critical timing for the input pair $DataA \rightarrow DataA$ is to be investigated, the algorithm would notice that a subsequent pulse on $DataA$ will not affect the current state after a fluxon is already stored in loop B_2—L_1—B_3 due to the effects of the first pulse on $DataA$. The algorithm would then erroneously assume that no critical time exists between subsequent applications of pulses on $DataA$. In terms of this example the current stability of the series junction B_1 needed to be considered as well. This method therefore does not take into account the effects of the storage or loss of a fluxon on other parts of the circuit.

Given the arguments thus far a possible generalisation could be stated as follows:

A critical timing relationship exists between two input pulses if the first input pulse causes the circuit to change state, and if the circuit behaves differently due to the application of the second pulse in the two respective states [47].

The input pair $DataA \rightarrow Clock$ fits the above generalisation for a critical timing relationship. The application of a pulse on $DataA$ causes the circuit to change state, and a subsequent application of a pulse on $Clock$ has a different effect on the circuit before and after the state change. More specifically the change in circuit behaviour is such that before the arrival of $DataA$, a $Clock$ pulse would cause B_4 to switch and after the application of $DataA$, a $Clock$ pulse causes B_3 and B_5 to switch.

The input pair $DataA \rightarrow DataA$ also fits the generalisation. The first pulse on $DataA$ causes the state to change due to the fluxon storage. The effects of the subsequent $DataA$ pulse differs before and after the state transition. In this case, B_2 switched before the state change while B_1 switched after the state transition.

The input pair $DataA \rightarrow DataB$ does not have a critical timing relationship which is also indicated by the generalisation. Even though a pulse on $DataA$ causes the circuit state to change, a subsequent pulse on $DataB$ has no different effect on the circuit before and after the state change.

Since the changes in the circuit are dictated by certain junctions in the circuit, a natural conclusion is to rather investigate the stability of these components in order to ascertain their critical timing relationships. For the input pair $DataA \rightarrow Clock$ the difference in elements are B_4, B_3 and B_5. Therefore the critical timing relationship is the time required for the last of these components to stabilise after the application of $DataA$.

Note that the effects of $Clock$ on the three critical junctions are not instantaneous after the application of the $Clock$ pulse itself. The voltage pulse has a certain delay before its effects are applied to the critical junctions. In the algorithm this time is also calculated and subtracted from the previously calculated critical time so as not to unnecessarily lengthen the established critical time.

A further refinement was also required that is not stated in [47]. It is also noteworthy that the generalisation does not hold for the input pair $Clock \rightarrow Clock$. If we start the application in State 0 there will be no change in the state after the first application of $Clock$. Hence, based on the generalisation, the algorithm will assume that there is no relationship between the input pairs. This, however, might not be the case since the application of two $Clock$ pulses with a very small delay might have functional effects on the circuit. It was therefore decided to rather test pulses that do not fit the generalisation by using the empirical timing characterisation method explained previously. For example, when the algorithm notices that the input pair does not fit the generalisation, a SPICE run is performed where the pair is applied to the circuit in the quickest possible succession (usually 1 ps delay). If the correct final state is reached by using this application, the algorithm indicates that no critical timing relationship exists between this input pulse pair. If the correct state was not reached, the empirical method needs to be applied to the input pair so as to determine the correct relationship, as explained previously.

Even with the refinements, it cannot be claimed that the method will work under all circumstances. It cannot, for example, be conclusively proven that junctions whose behaviour remains constant before and after a state change cannot have any functional effect due to transient effects on the circuit behaviour. The proposed method does, however, function robustly for current use cases.

An overview of the stability critical timing extraction algorithm is presented in Figure 5.10. At the start of the algorithm an untested input pair in an untested state is selected for investigation. The known state machine representation is used to ascertain whether the first pulse in the pair causes a state change in the system. If not, a shortest possible delay between the pulses is tested to identify a possible critical timing relationship. If a relationship is

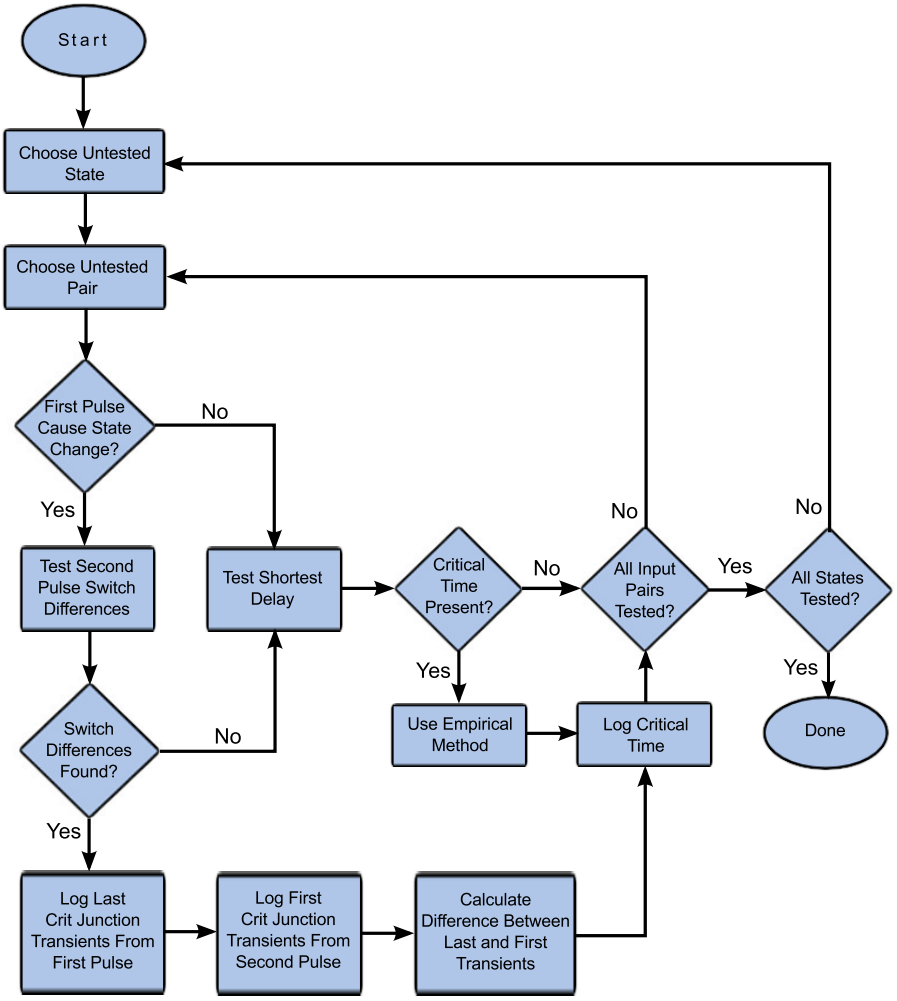


Figure 5.10: Current stability critical timing extraction algorithm.

identified, the empirical extraction method described previously is employed to calculate the critical time of the input pair. If the first pulse in the pair does in fact cause a state change, a SPICE run is used to ascertain whether the second pulse causes a difference in any junction switching between the two respective states. If no differences are identified, the empirical method is once again employed to test for a critical time relationship. If differences in the junction switches were found, the first pulse is applied to the circuit and the last transient time of all the critical junctions is logged. The second pulse is then used to log the first possible effect of the input on the critical junctions (in both states reached). The difference between the two times are then logged as the critical time relationship between the input pair. The algorithm is completed after all the input pairs for all the states have been investigated.

```
p_time : PROCESS (in_A, in_B) IS
BEGIN
  -- Investigate for violations
  IF rising_edge(in_A) AND (
   s0_in_B_in_A_err = '1' OR
   s1_in_A_in_A_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(in_B) AND (
   s1_in_A_in_B_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;
  -- Update violation signals
  CASE current_state IS
    WHEN state_0 =>
      IF rising_edge(in_B) THEN
        s0_in_B_in_A_err = TRANSPORT '1', '0' AFTER 42 PS;
      END IF;
    WHEN state_1 =>
      IF rising_edge(in_A) THEN
        s1_in_A_in_A_err = TRANSPORT '1', '0' AFTER 5 PS;
      END IF;
      IF rising_edge(in_B) THEN
        s1_in_A_in_B_er = TRANSPORT '1', '0' AFTER 45 PS;
      END IF;
  END CASE;
END PROCESS;
```

Figure 5.11: Example error verification VHDL code

5.4.6 HDL Generation

Both the state machine representation and the critical timing information is necessary for the generation of an HDL model. There is no limitation as to which HDL modelling language should be used for the model generation step since all the commonly used languages (VHDL, Verilog, System Verilog etc.) contain the necessary constructs for the behavioural and timing implementation. Currently the software library only allows for the generation of VHDL models although other language support can be added in future.

Since VHDL (and other HDL languages) primarily make use of voltage state logic, the presense of an SFQ pulse is modelled as a picosecond long pulse. The rising edge of this pulse is used to trigger the sequential HDL logic. The implementation of the critical timing checks are discussed in relation to Figure 5.11.

Every critical timing relationship has an associated critical time signal. If the signal value is a logical 1, this signifies that the arrival of the second pulse in the critical time pair will result in a timing violation. In the example code, the circuit being modelled has two inputs (in_A and in_B). The critical timing extraction method identified three relationships. The first and second relationship is $in\_B \rightarrow in\_A$ and $in\_A \rightarrow in\_A$ when in State 0, and $in\_A \rightarrow in\_B$ when in State 1. The associated error signals are $s0\_in\_B\_in\_A\_err$, $s0\_in\_A\_in\_A\_err$ and $s1\_in\_A\_in\_B\_err$ re-

spectively. The dynamics of the error signals are modelled by using VHDL TRANSPORT delays. This ensures that the error signal returns to a logical 0 after the critical timing period has passed. The updating of the error signals are implemented by means of a CASE construct.

The implementation of the state machine makes use of standard VHDL methods to implement sequential logic and will not be elaborated on.

5.4.6.1 HDL Generation Example

A reconfigurable Destructive-Flip-Flop(DFF)-JTL circuit was selected to demonstrate the HDL generation methodology. The circuit schematic is illustrated in Figure 5.12 with component parameters available in Table 5.1. The shunt resistors were removed for the sake of clarity. All input pulses are applied through a DC-SFQ circuit connected to the circuit under test using a Josephson Transmission Line while the outputs are terminated using a 2Ω resistor connected to the circuit also through a Josephson Transmission Line.

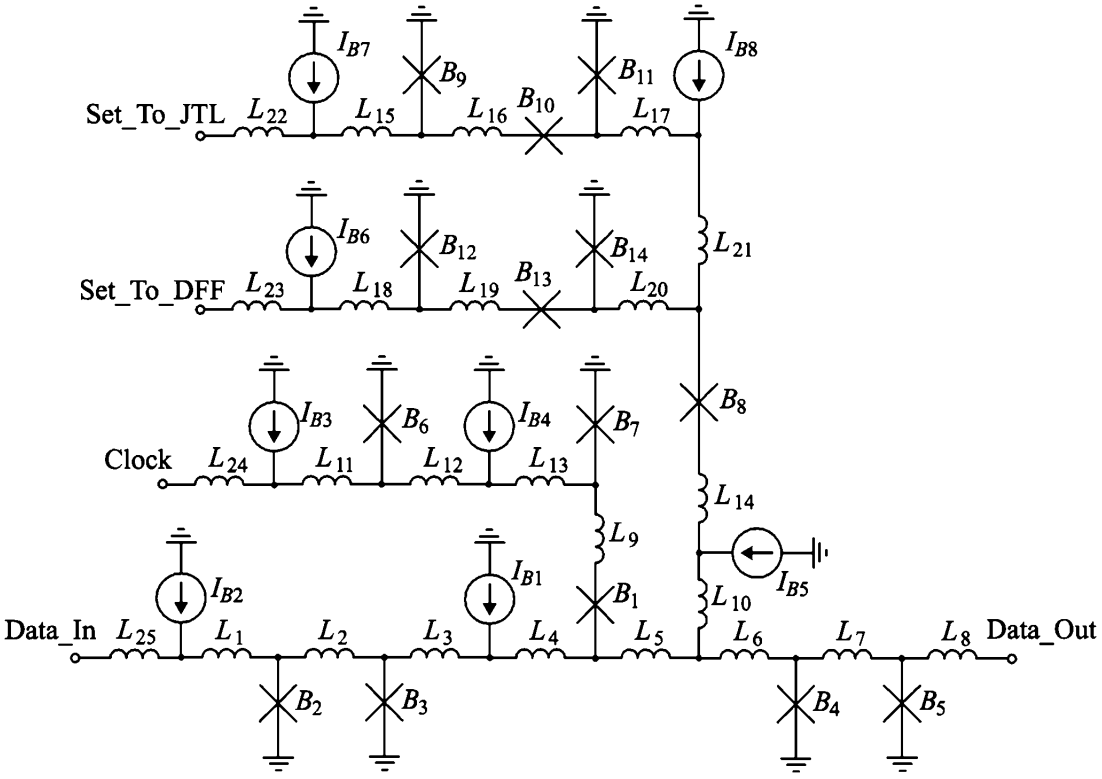


Figure 5.12: Schematic circuit of an unoptimised reconfigurable DFF-JTL

The circuit can function either as a D-Flip-Flop or a JTL depending on which of the two set-up inputs, Set_To_JTL or Set_To_DFF, last received an input pulse. At start-up, the circuit functions as a D-Flip-Flop due to a lack of a fluxon being stored in the loop B_{14}—L_{20}—L_{21}—L_{17}—B_{11}. The lack of a

Junctions	Inductors	Bias Currents
$B_1 : 100\,\mu m^2$	$L_1 : 2.5\,pH$	$I_{b1} : 81\,\mu A$
$B_2 : 200\,\mu m^2$	$L_2 : 2\,pH$	$I_{b2} : 300\,\mu A$
$B_3 : 100\,\mu m^2$	$L_3 : 2.2\,pH$	$I_{b3} : 300\,\mu A$
$B_4 : 100\,\mu m^2$	$L_4 : 2\,pH$	$I_{b4} : 150\,\mu A$
$B_5 : 225\,\mu m^2$	$L_5 : 2.1\,pH$	$I_{b5} : 63\,\mu A$
$B_6 : 200\,\mu m^2$	$L_6 : 0.1\,pH$	$I_{b6} : 300\,\mu A$
$B_7 : 300\,\mu m^2$	$L_7 : 4.6\,pH$	$I_{b7} : 300\,\mu A$
$B_8 : 100\,\mu m^2$	$L_8 : 1.8\,pH$	$I_{b8} : 104\,\mu A$
$B_9 : 200\,\mu m^2$	$L_9 : 2\,pH$	
$B_{10} : 200\,\mu m^2$	$L_{10} : 0.5\,pH$	
$B_{11} : 125\,\mu m^2$	$L_{11} : 2.5\,pH$	
$B_{12} : 200\,\mu m^2$	$L_{12} : 2.5\,pH$	
$B_{13} : 225\,\mu m^2$	$L_{13} : 0.1\,pH$	
$B_{14} : 250\,\mu m^2$	$L_{14} : 0.5\,pH$	
	$L_{15} : 2.3\,pH$	
	$L_{16} : 3.9\,pH$	
	$L_{17} : 0.8\,pH$	
	$L_{18} : 2,5\,pH$	
	$L_{19} : 2.9\,pH$	
	$L_{20} : 2.9\,pH$	
	$L_{21} : 3.5\,pH$	
	$L_{22} : 1.8\,pH$	
	$L_{23} : 1.8\,pH$	
	$L_{24} : 1.8\,pH$	
	$L_{25} : 1.8\,pH$	

Table 5.1: DFF-JTL circuit parameters.

stored fluxon influences the bias current through B_4, which in turn affects the behaviour of input pulses on $Data\_In$. An input pulse on $Data\_In$ causes a fluxon to be stored in the loop B_3—L_3—L_4—B_1—L_9—B_7. A subsequent input pulse on the $Clock$ input will cause B_4 to switch allowing the stored fluxon to propogate through $Data\_Out$. If no $Data\_In$ pulse arrived prior to the $Clock$ arrival, the $Clock$ pulse is removed from the circuit by the switching of B_1. This circuit demonstrates the fluxon storage exception that was explained previously. Normally the stored fluxon in loop B_3—L_3—L_4—B_1—L_9—B_7 would cause a shielding current through L_5, L_6 and B_4 to counteract the net flux gain in loop B_3—L_3—L_4—L_5—L_6—B_4 . The size of B_4 is, however, relatively small and this shielding current would cause B_4 to exceed its critical current. As explained earlier, this will cause some of the shielding current to rather shunt through L_7 and B_5. In this case, the extraction algorithm either indicates that B_3—L_3—L_4—B_1—L_9—B_7 is a fluxon storing inductive loop, or that the combination of B_3—L_3—L_4—L_5—L_6—B_4 and B_4—L_7—B_5 are used

for the storage. Both of these choices will result in the correct state machine representation.

An input on Set_To_JTL will have the effect of storing a fluxon in B_{14}—L_{20}—L_{21}—L_{17}—B_{11} and therefore also an associated effect on the bias current of B_4. Any subsequent inputs on $Data\_In$ will now rather switch B_4 before fluxon storage can occur. This will then propagate the pulse to $Data\_Out$.

If the circuit is in the D-Flip-Flop mode with a fluxon being stored, an input on Set_To_JTL will cause the stored fluxon to be propagated to $Data\_Out$ before returning to JTL mode.

The $Data\_In$ input does not have a series junction to guard against two subsequent pulses before a $Clock$ pulse arrives. If this should happen, the second $Data\_In$ pulse will cause B_4 to switch without the need of a $Clock$ pulse. In order to indicate to the algorithm that this situation should not be investigated, a user defined rule is necessary. The appropriate rule is indicated in equation 5.4.5 where node 17 is $Data\_In$, node 16 is $Clock$, node 15 is Set_To_DFF and node 14 is Set_To_JTL.

$$rule\ max\ 17\ 1\ after\ 15\ reset\ 16\ override\ 14 \tag{5.4.5}$$

The rule states that a maximum of one pulse can be applied to node 17. This rule is only valid when the "after" node receives an input after the "override" node. In this case, the rule is only valid when the Set_To_DFF pulse arrives after the Set_To_JTL pulse. The rule is reset after the arrival of a pulse on node 16.

All the inputs of the circuit-under-test were connected to the output of a DC-SFQ and JTL pair in order to generate a representative SFQ pulse from a piece-wise linear SPICE input. The output was connected to a JTL before terminating in a 2Ω resistor.

The extracted state machine representation is illustrated in Figure 5.13 which was reproduced from [47].

The system has two fluxon storage loops which equates to a maximum of four states. However, only three states are reachable since both loops never store a fluxon simultaneously. State 0 is associated with the DFF functionality where no fluxon is stored in B_{14}—L_{20}—L_{21}—L_{17}—B_{11}. State 2 is associated with the storage of a fluxon in B_{14}—L_{20}—L_{21}—L_{17}—B_{11} after the arrival of a pulse on $Data\_In$. State 1 is associated with the storage of a fluxon in B_{14}—L_{20}—L_{21}—L_{17}—B_{11}. The circles on the transition arrows indicate when an output pulse is generated. It is important to note that an additional $Data\_In$ pulse in State 2 is not allowable due to the lack of a series junction in the input path.

After the state machine is generated the critical timings of the circuit are extracted before the VHDL model is constructed. For the purposes of comparison both of the critical timing extraction methods were applied to the

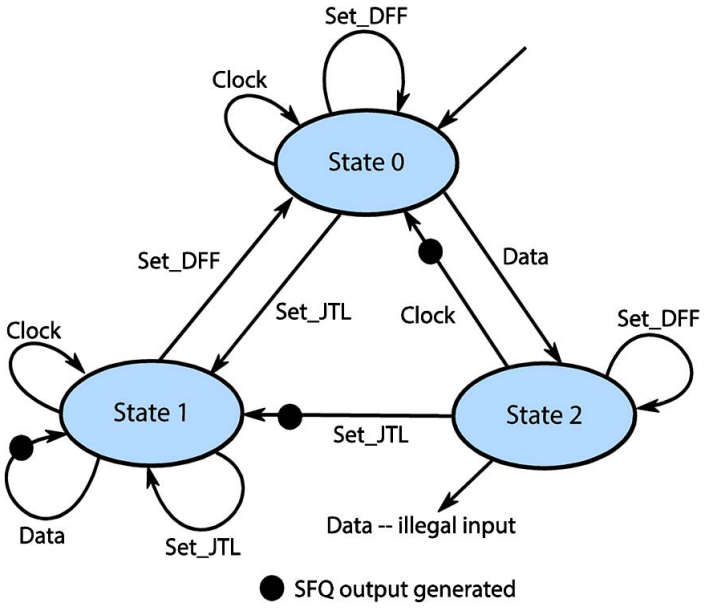


Figure 5.13: Extracted DFF-JTL Mealy state machine representation.

DFF-JTL circuit. The VHDL error code for the empirical extraction method is shown in Figure 5.14 where the allowable timing uncertainty was selected as 4 ps. The critical timings by using the current stability method is shown in Figure 5.15 and in Figure 5.16.

The inputs of the VHDL model is associated with the input junctions that were used to calculate timing parameters. The input junctions were selected as the last junctions in the connected DC-SFQ-JTL pair. In the model, $X6\_B2$ is associated with the Set_To_JTL input, $X7\_B2$ is associated with the Set_To_DFF input, $X8\_B2$ is associated with the $Clock$ input and $X9\_B2$ is associated with $Data\_In$. Entity and architectural definitions were omitted for the sake of clarity.

From the figures presented above, it is evident that there is a substantial difference in the models of the two methods. The current stability method always errs on the side of caution since it is clear that many of the critical timings generated due to overlapping current dynamics do not empirically cause a circuit failure. One could argue, however, that the stability method results in a far more robust circuit. The stability method can further be refined to rather change the time selected for the start and end of the current transients to a certain percentage of the maximum current change.

The state machine implementation itself is equivalent between the methods and is presented in Figure 5.17.

VHDL and SPICE simulation results are illustrated in figures 5.18 and 5.19 which were reproduced from [47] to illustrate the equivalence between the SPICE circuit and the generated HDL model.

```
p_time : PROCESS (in_X8_B2, in_X6_B2, in_X7_B2) IS
BEGIN
  IF rising_edge(X8_B2) AND (s0_X9_B2_X8_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(X6_B2) AND (
   s0_X7_B2_X6_B2_err = '1' OR
   s2_X6_B2_X6_B2_err = '1' OR
   s2_X7_B2_X6_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(X7_B2) AND (
   s1_X6_B2_X7_B2_err = '1' OR
   s1_X9_B2_X7_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  CASE current_state IS
    WHEN state_0 =>
      IF rising_edge(X7_B2) THEN
        s0_X7_B2_X6_B2_err = TRANSPORT '1', '0' AFTER 8 PS;
      END IF;
      IF rising_edge(X9_B2) THEN
        s0_X9_B2_X8_B2_err = TRANSPORT '1', '0' AFTER 8 PS;
      END IF;
    WHEN state_1 =>
      IF rising_edge(X6_B2) THEN
        s1_X6_B2_X7_B2_err = TRANSPORT '1', '0' AFTER 5 PS;
      END IF;
      IF rising_edge(X9_B2) THEN
        s1_X9_B2_X7_B2_err = TRANSPORT '1', '0' AFTER 10 PS;
      END IF;
    WHEN state_2 =>
      IF rising_edge(X6_B2) THEN
        s2_X6_B2_X6_B2_err = TRANSPORT '1', '0' AFTER 45 PS;
      END IF;
      IF rising_edge(X7_B2) THEN
        s2_X7_B2_X6_B2_err = TRANSPORT '1', '0' AFTER 12 PS;
      END IF;
  END CASE;
END PROCESS;
```

Figure 5.14: Error verification VHDL code for the DFF-JTL circuit using the empirical extraction method.

```
p_time : PROCESS (in_X8_B2, in_X6_B2, in_X7_B2, in_X9_B2) IS
BEGIN
  IF rising_edge(X8_B2) AND (
   s0_X9_B2_X8_B2_err = '1' OR
   s2_X6_B2_X8_B2_err = '1' OR
   s2_X8_B2_X8_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(X6_B2) AND (
   s0_X6_B2_X6_B2_err = '1' OR
   s0_X7_B2_X6_B2_err = '1' OR
   s0_X9_B2_X6_B2_err = '1' OR
   s1_X7_B2_X6_B2_err = '1' OR
   s2_X6_B2_X6_B2_err = '1' OR
   s2_X7_B2_X6_B2_err = '1' OR
   s2_X8_B2_X6_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(X7_B2) AND (
   s0_X6_B2_X7_B2_err = '1' OR
   s1_X6_B2_X7_B2_err = '1' OR
   s1_X7_B2_X7_B2_err = '1' OR
   s1_X9_B2_X7_B2_err = '1' OR
   s2_X6_B2_X7_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;

  IF rising_edge(X9_B2) AND (
   s0_X6_B2_X9_B2_err = '1' OR
   s1_X7_B2_X9_B2_err = '1' OR
   s2_X6_B2_X9_B2_err = '1' OR
   s2_X8_B2_X9_B2_err = '1') THEN
    REPORT "Timing Violation" SEVERITY ERROR;
  END IF;
```

Figure 5.15: Error verification VHDL code for the DFF-JTL circuit using the current stability method.

```
    CASE current_state IS
      WHEN state_0 =>
        IF rising_edge(X6_B2) THEN
          s0_X6_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 42 PS;
        END IF;
        IF rising_edge(X6_B2) THEN
          s0_X6_B2_X7_B2_err <= TRANSPORT '1', '0' AFTER 43 PS;
        END IF;
        IF rising_edge(X6_B2) THEN
          s0_X6_B2_X9_B2_err <= TRANSPORT '1', '0' AFTER 20 PS;
        END IF;
        IF rising_edge(X7_B2) THEN
          s0_X7_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 8 PS;
        END IF;
        IF rising_edge(X9_B2) THEN
          s0_X9_B2_X8_B2_err <= TRANSPORT '1', '0' AFTER 46 PS;
        END IF;
        IF rising_edge(X9_B2) THEN
          s0_X9_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 49 PS;
        END IF;
      WHEN state_1 =>
        IF rising_edge(X6_B2) THEN
          s1_X6_B2_X7_B2_err <= TRANSPORT '1', '0' AFTER 5 PS;
        END IF;
        IF rising_edge(X7_B2) THEN
          s1_X7_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 45 PS;
        END IF;
        IF rising_edge(X7_B2) THEN
          s1_X7_B2_X7_B2_err <= TRANSPORT '1', '0' AFTER 48 PS;
        END IF;
        IF rising_edge(X7_B2) THEN
          s1_X7_B2_X9_B2_err <= TRANSPORT '1', '0' AFTER 25 PS;
        END IF;
        IF rising_edge(X9_B2) THEN
          s1_X9_B2_X7_B2_err <= TRANSPORT '1', '0' AFTER 10 PS;
        END IF;
      WHEN state_2 =>
        IF rising_edge(X6_B2) THEN
          s2_X6_B2_X8_B2_err <= TRANSPORT '1', '0' AFTER 77 PS;
        END IF;
        IF rising_edge(X6_B2) THEN
          s2_X6_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 82 PS;
        END IF;
        IF rising_edge(X6_B2) THEN
          s2_X6_B2_X7_B2_err <= TRANSPORT '1', '0' AFTER 86 PS;
        END IF;
        IF rising_edge(X6_B2) THEN
          s2_X6_B2_X9_B2_err <= TRANSPORT '1', '0' AFTER 62 PS;
        END IF;
        IF rising_edge(X7_B2) THEN
          s2_X7_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 12 PS;
        END IF;
        IF rising_edge(X8_B2) THEN
          s2_X8_B2_X8_B2_err <= TRANSPORT '1', '0' AFTER 66 PS;
        END IF;
        IF rising_edge(X8_B2) THEN
          s2_X8_B2_X6_B2_err <= TRANSPORT '1', '0' AFTER 65 PS;
        END IF;
        IF rising_edge(X8_B2) THEN
          s2_X8_B2_X9_B2_err <= TRANSPORT '1', '0' AFTER 41 PS;
        END IF;
    END CASE;
  END PROCESS;
```

Figure 5.16: Error verification VHDL code for the DFF-JTL circuit using the current stability method continued.

```
BEGIN
  CASE current_state IS
    WHEN state_0 =>
      IF rising_edge(in_X6_B2) THEN
        current_state <= state_1;
      END IF;
      IF rising_edge(in_X9_B2) THEN
        current_state <= state_2;
      END IF;
    WHEN state_1 =>
      IF rising_edge(in_X7_B2) THEN
        current_state <= state_0;
      END IF;
      IF rising_edge(in_X9_B2) THEN
        out_X11_B1_sig <= TRANSPORT '1' AFTER 43 ps, '0' AFTER 44 ps;
      END IF;
    WHEN state_2 =>
      IF rising_edge(in_X8_B2) THEN
        current_state <= state_0;
        out_X11_B1 <= TRANSPORT '1' AFTER 34 ps, '0' AFTER 35 PS;
      END IF;
      IF rising_edge(in_X6_B2) THEN
        current_state <= state_1;
        out_X11_B1 <= TRANSPORT '1' AFTER 43 ps, '0' AFTER 44 PS;
      END IF;
      IF rising_edge(in_X9_B2) THEN
        REPORT "Illegal Input Sequence" SEVERITY ERROR;
      END IF;
  END CASE;
```

Figure 5.17: VHDL state machine implementation of the DFF-JTL circuit.

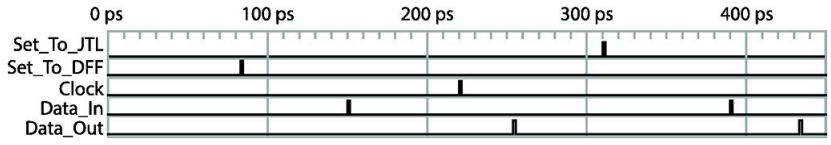


Figure 5.18: Functional waveforms of DFF-JTL VHDL simulation.

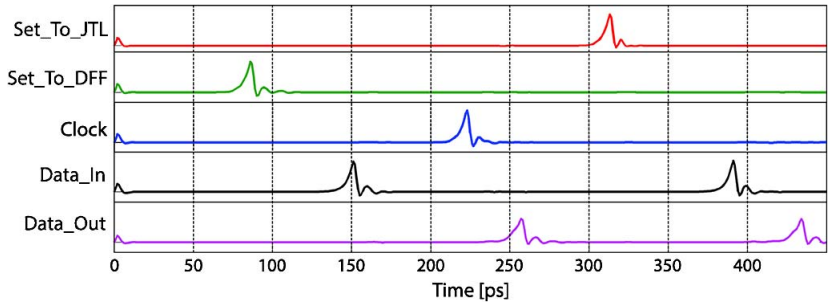


Figure 5.19: SPICE transient simulation of DFF-JTL circuit.

Chapter 6

Yield Optimisation of RSFQ Circuits

6.1 Introduction

Optimisation is the minimisation (or maximisation) of a function given certain variable constraints. Mathematically, optimisation can be described as follows [50]:

$$\underset{x \in \mathcal{R}^n}{\text{minimise}} \quad f_i(\mathbf{x}), \quad (i = 1...M) \tag{6.1.1}$$

$$with \quad \phi_j(\mathbf{x}) = 0 \quad (j = 1...J) \tag{6.1.2}$$

$$\upsilon_k(\mathbf{x}) \leq 0 \quad (j = 1...K) \tag{6.1.3}$$

In equations 6.1.1, x is the input vector (also called the design vector) and f_i is the function to be minimised(maximised), also known as the cost or merit function. The subscript indicates that more than one objective can be used in the optimisation process. The constraints can generally be classified as equalities (for ϕ) and bounds (for υ) which are applicable to the input vector. In the equation, the input vector is designated as being in the set of real numbers. The equations are, however, applicable to integer sets as well.

In this section the main focus falls on the optimisation of circuit yield.

Circuit yield is the percentage of manufactured circuits that are classified as functional, given their designer specifications and manufacturing process variations. Various methods of circuit yield calculations were investigated in Chapter 3 of this study.

The need for optimisation in terms of circuit yield is self-evident. Commercially speaking, less circuit failures equate to more profit. In terms of research, more robust circuits equate to the possible design of more complex circuitry.

In the semiconductor industry, yield optimisation (also known as design centering) is still an active research area, even after decades of maturity and

experience in the manufacturing of such circuits. The reason for this appears to be the constant shrinking of node sizes with associated changes in the manufacturing process.

There is no reason to believe that this will be any different for RSFQ circuits (or for any other next-generation digital logic circuitry). RSFQ circuit design is still in its infancy, even after a few decades of active research. This includes manufacturing processes that cannot be compared with the complexity of processes used in the semiconductor industry. In order to stimulate research and funding in the superconducting digital logic field, complex and useful circuits need to be designed, shown to be functional, shown to be comparable and, hopefully, superior to their semiconductor counterparts. Yield optimisation forms an integral part of this process to create complex, robust circuits.

Substantial research has already been conducted in the field of RSFQ yield optimisation. Many of these methods were inspired by the semiconductor industry, given the overlapping similar problem of process spread. Currently, the optimisation methods investigated for RSFQ circuits fall into two categories: geometric and heuristic. Geometric methods attempt to model the feasible area in the possible search space by using multi-dimensional geometric structures. The feasible area is the area in the search space where the circuit is deemed functional. Once a geometric model is generated, a mathematical method can be used to find the centre of this structure, which then indicates the optimal design. It follows that the highest yield can be found at this point since it allows for the maximum amount of deviation in all dimensions while still maintaining the functionality of the circuit. An example of this method using hyper-spheres can be found in [51].

The second type of optimisation that is employed to maximise circuit yield is heuristic and meta-heuristic methods. A heuristic can be thought of as an experience-based technique that can be applied to solve a particular problem. A meta-heuristic is a technique that can be applied to many different types of problems. Generally, heuristics do not guarantee optimal solutions, but rather tend to produce good solutions for difficult optimisation problems. A staple of modern heuristics is also a random element which tends to make any rigorous mathematical proof of convergence a very difficult undertaking. On the other hand, it allows these methods to be used to solve problems that cannot be approached analytically at all. Some of these methods can be found in [52, 53, 54]. As is evident, many of these heuristics are inspirated by natural phenomena in order to address the optimisation problem.

Comparing heuristics is a difficult task due to their intrinsic random nature. One can, for example, do a few runs of each heuristic and illustrate the final merit value that is achieved. However, these results might never be repeatable due to the random nature of the algorithms. A more robust approach for comparing this probabilistic data would be to perform many runs and obtain statistical information on the data provided. This implies an investigation into

the best merit achieved, the mean of all the runs, the standard deviation of the runs, and so on, in order to ascertain whether there really is an improvement of any particular algorithm over another. The problem is exacerbated by the amount of time it takes to perform an optimisation run for a single, very small circuit. Depending on the confidence level of the requires values, an optimisation run can take anything from a few minutes to a few days which renders the gathering of vast amounts of data very time-consuming.

The alternative is to use a model of the merit space that can be sampled in a fraction of the time a SPICE run would take. This ability is provided by the artificial neural networks discussed in Chapter 4. Once the network has been trained it can be reused for the testing of all the heuristics. Bearing in mind though that the training of an artificial neural network is not an exact science. One might, for example, miss subtle behaviours in the merit function either because no training data were available at those locations, or the network parameters were not set up correctly to incorporate the correct behaviour. Either way, as is presented below, the networks will serve as an approximation of the shape of the search space which serves as a tool to test optimisation runs thousands of times in order to extract statistical data.

In this section three heuristics are investigated. These are the greedy local search, simulated annealing and genetic algorithm. All three heuristics are investigated by means of artificial neural networks so as to determine their merit function approximation.

The presented optimisation algorithms form part of the software library described in Appendix A, which can easily be extended to include other algorithms.

The algorithms are presented in the next section, along with their parameters that will be varied in order to investigate their behaviour. The RSFQ circuits that form part of the investigation are then presented along with an investigation of their yield search spaces. The solution representation is then explained after which the results of the optimisation algorithms are presented along with a conclusion.

6.2 Meta-Heuristics

6.2.1 Greedy Local Search

Greedy Local Search [55] is a very simple randomised heuristic that serves as a control for comparison purposes. It can be classified as a local search methodology. It commences with a single solution and steps through the neighbourhood using random perturbations of this solution. A flow diagram of the algorithm is shown in Figure 6.1.

The algorithm is initiated by creating a viable solution situated in the feasible region of the search space. The merit value of the solution is then

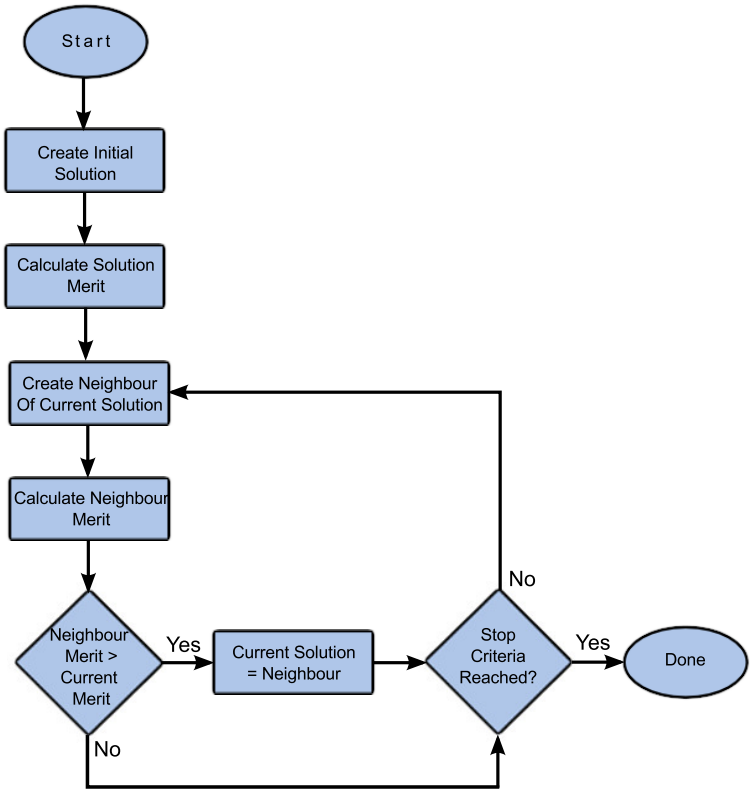


Figure 6.1: Greedy local search algorithm flow diagram.

calculated using the chosen merit function. With this initial solution, a viable neighbouring solution is then selected by randomly varying the value of a component (or multiple components). What exactly constitutes the neighbourhood is explored later in this section. The merit value of the neighbouring solution is then calculated and compared to that of the initial solution. If the merit value of the neighbouring solution is higher (for maximisation optimisation), then this solution is selected as the current solution and the initial solution is discarded. However, if the merit value of the neighbouring solution is lower than that of the initial solution, the neighbouring solution is discarded and the initial solution retained as the current solution. A new neighbour is then created from the current solution and compared in terms of merit value. The algorithm ends when a certain stop criteria has been reached. This criteria can generally be the number of tested solutions, an adequate merit value being reached or no merit increase being found in a certain number of neighbours generated.

The only variable that can influence the behaviour of a greedy local search algorithm is the neighbourhood definition of a solution. The neighbourhood is described in the variation length (or neighbourhood size) and the number of variable dimensions. The variation length designates the range (from the nominal solution value) from which the neighbouring solution can be selected.

For the solution representation, the scope can vary from one step (in a positive or negative direction) up to the whole range of the component, depending on the variation length parameter. Another possible variable may determine the number of dimensions (or components) across which perturbation should be allowed.

The choice of these parameters can have a profound effect on the workings of the algorithm. For example, by choosing 1 as the allowable step selection, the algorithm stands a very good chance of getting stuck in a local optimum (if one exists). However, if a large step size is selected, the algorithm could "jump" over a local optimum thus allowing a global optimum to be found. The selection of multiple variable dimensions also allows the algorithm to move to better solutions for problems that have a significant influence on the parameters.

The effects of the variation length (neighbourhood size) and variable dimensions (neighbourhood dimension) are investigated below.

6.2.2 Simulated Annealing

The simulated annealing algorithm takes its name from the annealing process in solids [56]. A crystalline solid is heated and slowly cooled to allow the crystal lattice configuration to reach its most regular (least energy) state. This leads to superior structural integrity of the solid. The basics of the optimisation algorithm itself is closely related to that of a local search algorithm (such as the Greedy Local Search). The algorithm commences with a viable solution from which a neighbouring solution is created. The difference lies in the acceptance criteria of this new solution. Unlike with a greedy local search, a worse solution can be selected for the current solution depending on a certain, variable probability. This probability is dependent on the difference in merit values of the two solutions and on a parameter called the temperature. The higher the temperature value, the more likely the algorithm will be to select a worse solution. The change in the temperature variable is governed by the cooling schedule of the algorithms, which is elaborated on below. The Metropolis acceptance criteria [56] is generally used to define the probability of selecting a worse solution. These criteria are presented in equation 6.2.1. The probability of selecting an inferior solution that has a yield value of 20% less than the comparative solution is depicted in Figure 6.2.

The acceptance criteria provide a possible method for the algorithm to avoid a local optimum by allowing for inferior steps. As illustrated in Figure 6.2, at the start of the run, when the temperature value is high, the probability of selecting an inferior solution is high too. As the temperature decreases, however, less and less inferior solutions are accepted. When the temperature variable reaches zero, the algorithm reduces to a simple greedy local search as explained earlier. Nonetheless, it is noticeable that this methodology does not guarantee that simulated annealing will always perform better than its greedy

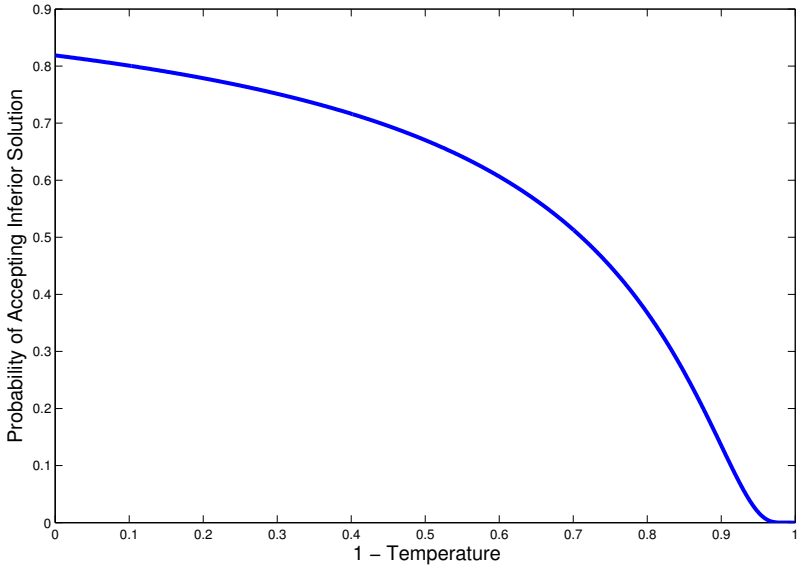


Figure 6.2: Graph of probability of selecting an inferior solution using simulated annealing.

local search counterpart. Simulated annealing only provides the mechanics for the algorithm to possibly settle near the global optimum before commencing a greedy search. The greedy local search algorithm, on the other hand, is completely dependent on its starting value, and the size and dimensionality of the defined neighbourhood.

$$P(Accept\ \ \omega') = \begin{cases} exp[(f(\omega') - f(\omega))/t_k] & \text{if } f(\omega') < f(\omega) \\ 1 & \text{if } f(\omega') \geq f(\omega) \end{cases} \tag{6.2.1}$$

It is interesting to note that simulated annealing is one of the few metaheuristics that indicate proof of convergence. The interested reader can investigate this convergence proof in [56].

The algorithm is further explained in Figure 6.3. It commences with the creation of an initial, random solution in the feasible region of the search space after which the merit value of this solution is calculated. A neighbouring solution is then constructed using the initial solution followed by the merit calculation of the neighbouring solution. If the merit value of the neighbouring solution is higher (given a maximisation problem) than that of the initial solution, the neighbouring solution is selected as the current solution. If the merit value of the neighbouring solution is lower than that of the initial solution, the acceptance criteria in equation 6.2.1 is used to either select the neighbouring solution as the new current solution, or to discard it depending on a sample from a uniformly distributed random variable.

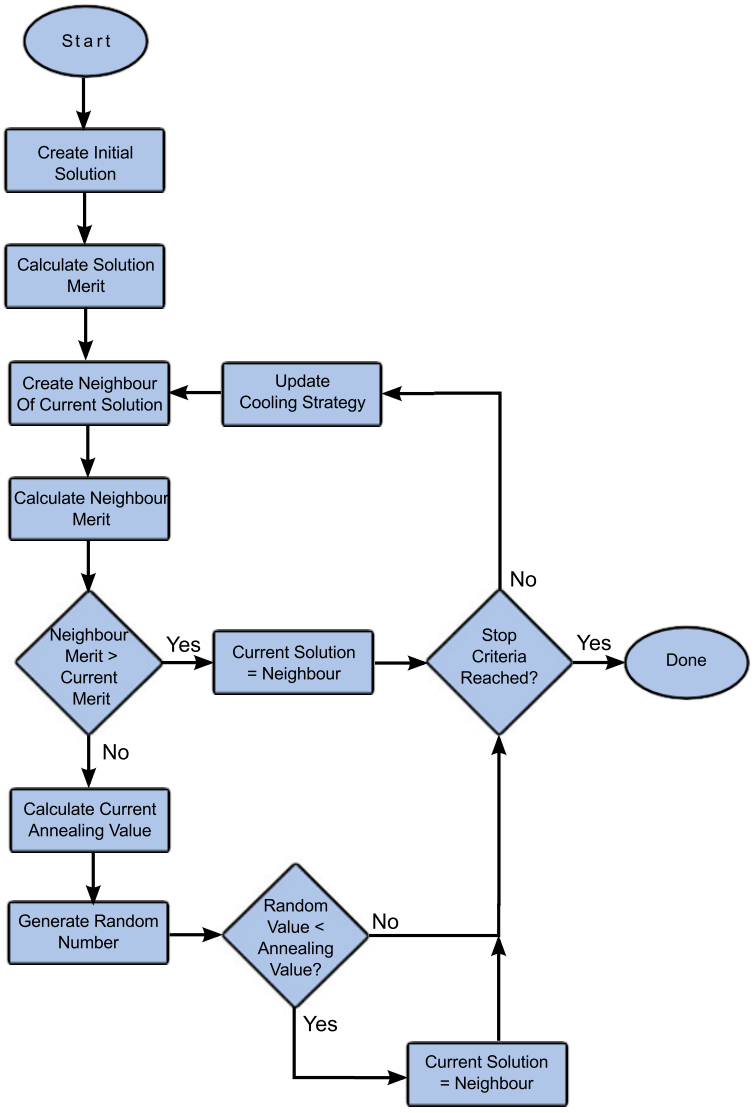


Figure 6.3: Simulated annealing algorithm flow diagram.

The stop criteria can be associated with a certain merit value, an absolute number of runs, or a stability criterion. Generally, the strategy allows the temperature variable to reduce to zero, after which a greedy implementation is run until no improvements are found for a certain number of samples. If the number of stable samples are large, the possibility of a local (or global) optimum being found is high.

The cooling strategy of the algorithm defines how the temperature variable varies as the algorithm progresses. This normally consists of maintaining the temperature variable at a stable rate for a number of steps after which the temperature is decreased with a certain step size. In general, the decrease in the temperature variable is linear or geometric. The cooling strategy is discussed in detail below.

Simulated annealing has the same neighbourhood definition as the greedy local search heuristic. The same neighbourhood parameters are therefore available as variables. These are the number of dimensions that can be perturbed when seeking a neighbouring solution and also the maximum step size that can be taken from the initial solution.

A unique feature of the simulated annealing algorithm is its cooling strategy. This strategy governs how the temperature variable changes as the algorithm progresses. For the purpose of heuristic comparison, a linear cooling strategy is used. This entails maintaining the temperature at a stable rate for a number of steps after which the temperature is reduced by a fixed step size. For this implementation three free variables are evident: the first variable is the starting temperature, the second is the number of iterations per temperature step and the last variable is the temperature step itself. As explained previously, the aim is to allow the algorithm to settle in an advantageous position within the search space before running a greedy implementation with a temperature variable that had reached zero.

It was decided to reduce the three variables to two by using a fixed allowable number of samples until the temperature reached zero. The number of steps that the cooling strategy can take from its starting point until it reaches zero, multiplied by the number of iterations per temperature step are held constant. This is realised to ensure that one set-up does not perform better than another because it was allowed to use more samples. For example, if 1000 samples are selected as a constant and 10 steps are chosen for the starting temperature value to reach zero, 100 iterations per temperature should be applied.

The free variables that are investigated for the simulated annealing algorithm reduce to the starting value and the number of temperature steps that the cooling strategy will take.

6.2.3 Genetic Algorithm

Genetic algorithms [57] are meta-heuristics inspired by the genetics of evolution, with the concept of survival of the fittest playing a large part in the algorithm. The notion behind this heuristic is that a fit (good) solution (genome) carries some features that are the cause of the fitness. This might, for example, either be the presence of a certain parameter (gene) value in the genome or the presence of a group of interrelated parameter values. The methodology is to attempt to pass these good features on to the next generation by "breeding" the solution with another (usually also good) genome. A mutation strategy usually also forms part of this heuristic. This entails changes to one or more structures of a single genome in order to add new genetic information into the generation that was not available in the gene pool before as well as to perform a rudimentary local search. An "elitist" strategy is also usually employed. This entails that a certain percentage of the best genomes in the current generation will be directly transferred to the new generation. This af-

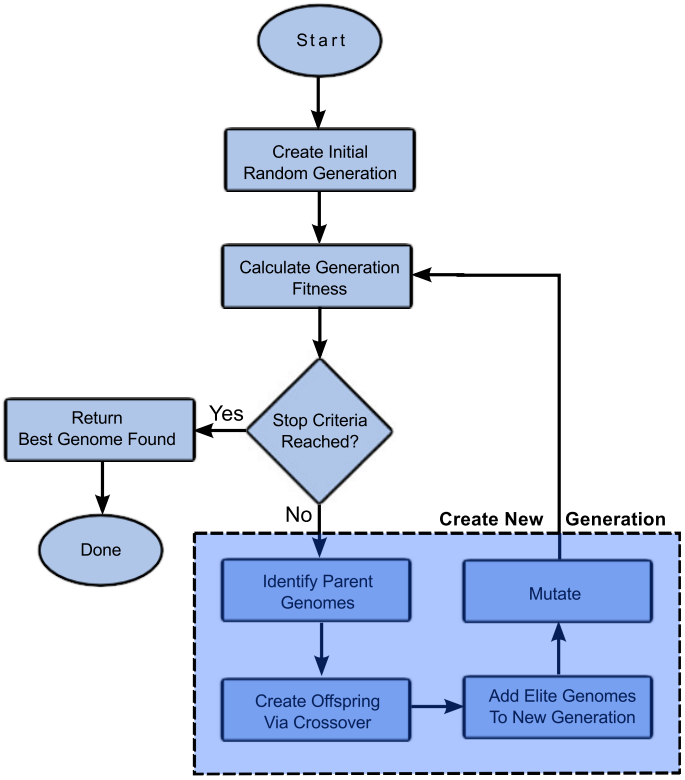


Figure 6.4: Genetic algorithm flow diagram.

fords a further opportunity to incorporate the good features of these genomes into future generations.

The algorithm is explained further in Figure 6.4.

The algorithm commences by creating an initial generation which consists of random, but usually viable genomes after which the fitness (merit value) of each genomes is calculated. After each fitness calculation process the stop criteria is checked, which is explained below.

If the stop criteria was not reached, a new generation of genomes is created from the current generation (illustrated by the shaded area in the figure). The composition of the new generation is variable based on the decision of the designer. Normally a percentage, called the crossover percentage, is used to define what percentage of the new generation should be created via "breeding" (or crossover as referred to in genetic algorithm literature). Before these new genomes can be constructed, a gene pool is first created. This is a list of the genomes in the current generation that are elected to take part in the crossover process. Generally this list contains the best individuals of the current generation. The percentage of the ordered list (best to worst) of individuals in the current generation that are ellected to take part in the crossover is called the threshold percentage. Finally the difference between the number of individuals in a generation and the number of individuals created by the crossover, is made up of the best individuals of the current generation thus incorporating an

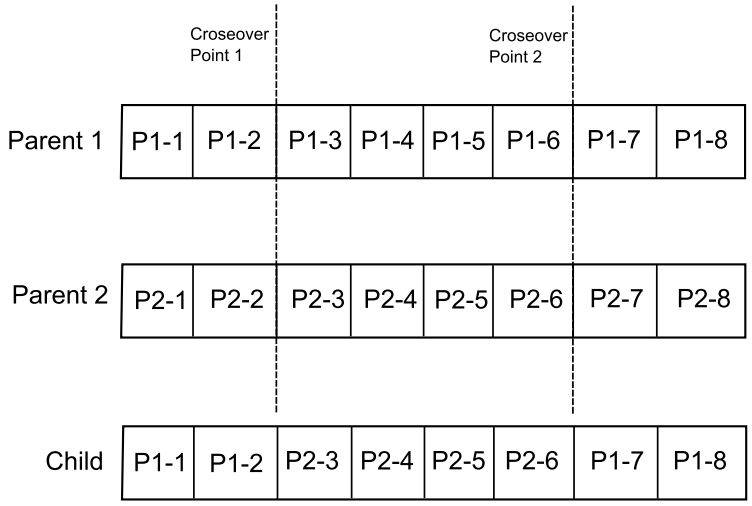


Figure 6.5: Genetic algorithm crossover illustration.

elitist strategy which ensures that the gene pool contains a sufficient amount of good genomes.

The stop criteria of the algorithm can once again either be the number of runs (or the number of generations in this case), a certain merit (fitness) value being reached, or a certain number of runs for which no improvements were found.

Genetic algorithms are easy to implement but hard to use optimally. This is due to the number of variable parameters in the algorithm that have an effect on its behaviour. The inherent power of the algorithm emanates from its parallel implementation. In genetic algorithms, unlike with greedy local search and simulated annealing algorithms a large diverse area of the search space is investigated in parallel which improves the chances of identifying promising search areas. The way in which these areas are searched is governed by the parameters of the algorithm which are discussed below.

The first parameter of interest is the size of the generation. This is the number of individuals in each generation. It is generally favourable to have a large generation size as this ensures that various parts of the search space is represented in the generation. This does, however, add substantially to the processing time necessary to calculate the generational fitness.

The next significant parameters are related to how any subsequent generation is constructed. The *truncation percentage* is the percentage of the current generation that will form part of the gene pool. This is the list of viable parents for the crossover process. The next parameter of interest is the *crossover percentage*. This is the percentage of a new generation that should be created using the crossover operator.

The crossover itself can be defined in many different ways. The implemented crossover strategy is illustrated in Figure 6.5

A number of *crossover points* are defined for the algorithm which map to

indices in the solution representation. In the figure, each block of a solution corresponds to a parameter (for example a circuit component) and the value inside the block is the current value of the component in the allowable range. The crossover points therefore map to indices of the solution representation. The user of the algorithm only defines the number of crossover points and from then onwards the points themselves are randomly selected. The crossover process then steps through the solution representation assigning gene values from the parent to the child. At the start of the process, the gene values of parent 1 will be assigned to the child. As soon as a crossover point is reached, the source of points is switched to another parent, who then continues to assign values. When another crossover point is reached the source switches back to the first parent, and so on. In this manner there is a possibility that components that function well together, stay together in the genome of the child.

The last parameter of interest is the *mutation ratio*. This relates to the probability that an individual in the newly created generation will undergo mutation. The mutation process itself is akin to the neighbourhood search methods explained earlier. A number of component points (called dimensions previously) are selected along with a range over which these dimensions may vary. These components are then randomly assigned new values in the range. The same effects caused by the neighbourhood definition explained earlier are prevalent at the mutation stage.

In order to keep the investigation tractable, three parameters were selected to vary. These are the generation size, the number of crossover points and the mutation ratio. The generation size should in general have a significant effect on the algorithm given that this would increase the diversification and therefore increase the opportunity to find a good solution. The number of crossover points could also make an interesting contribution to the algorithm since the components in a circuit generally tend to have some correlation with regards to their values. The number of crossover points influence the amount of neighboring components that will remain together throughout the crossover process. The mutation ratio controls the localised search capability of the algorithm, and is therefore also important.

6.3 Circuits

The RSFQ AND and OR circuits were investigated for yield optimisation. These circuits were already introduced in earlier chapters in this document and are added here again for easy reference. Values for the junction capacitances and shunt resistances were obtained using the IPHT 1 kA process [49]. The respective values are listed in Table 6.1. All the shunt resistances were calculated for a β_c value of approximately 1. It is important to note that for the optimisation investigation smaller step sizes were used than those presented

in Table 6.1. A simple linear interpolation method was used to generate capacitance and shunt resistance values that are not shown. All input pulses are applied through a DC-SFQ circuit connected to the circuit under test using a Josephson Transmission Line while the outputs are terminated using a 2Ω resistor connected to the circuit also through a Josephson Transmission Line.

Junction Area	Junction Capacitance	Shunt Resistance
$100\,\mu m^2$	$0.498988\,pF$	$2.576\,\Omega$
$125\,\mu m^2$	$0.624488\,pF$	$2.057\,\Omega$
$150\,\mu m^2$	$0.785522\,pF$	$1.695\,\Omega$
$175\,\mu m^2$	$0.883018\,pF$	$1.456\,\Omega$
$200\,\mu m^2$	$1.00852\,pF$	$1.274\,\Omega$
$225\,\mu m^2$	$1.13151\,pF$	$1.135\,\Omega$
$250\,\mu m^2$	$1.26153\,pF$	$1.022\,\Omega$
$275\,\mu m^2$	$1.38502\,pF$	$0.9278\,\Omega$
$300\,\mu m^2$	$1.50550\,pF$	$0.8529\,\Omega$
$325\,\mu m^2$	$1.62999\,pF$	$0.7879\,\Omega$
$350\,\mu m^2$	$1.75851\,pF$	$0.7308\,\Omega$
$375\,\mu m^2$	$1.87848\,pF$	$0.6842\,\Omega$
$400\,\mu m^2$	$2.01704\,pF$	$0.6371\,\Omega$
$425\,\mu m^2$	$2.15234\,pF$	$0.5934\,\Omega$
$450\,\mu m^2$	$2.28123\,pF$	$0.5435\,\Omega$

Table 6.1: IPHT 1 kA junction parameters.

The AND gate in Figure 6.6 consists of two fluxon storage loops that are used to change the behaviour of the circuit. These loops are B_2—L_2—B_3 and B_{13}—L_{12}—B_{14}. Inputs on *DataA* and *DataB* are respectively stored in these loops. A pulse on *Clock* causes both of these stored fluxons to be released and, due to the biasing of B_6 and B_{10} causes B_8 to switch, thereby producing an output pulse. The storage of the fluxons influences the biasing of B_6 and B_{10}. If only one loop holds a fluxon, an arriving clock pulse will cause B_6 or B_{10} to switch before B_8, therefore no output pulse is produced. Junctions B_1 and B_{12} serve as escape junctions for consecutive data pulses without an interleaved *Clock* pulse. Junctions B_5 and B_9 form a pulse splitter for the application of the *Clock* pulse to the storage loops while junctions B_4 and B_{11} are used as escape junctions for the *Clock* pulses if no fluxons are present in the respective loops. The circuit parameters are again listed in Table 6.2.

It is also important to note that the two input data arms of the AND gate are identical. It is therefore unnecessary to optimise the components of these two arms independently since their parameter values will generally be identical. This information can be passed to the optimisation algorithm via a linked component list.

The next circuit under test is the RSFQ OR gate, depicted in Figure 6.7.

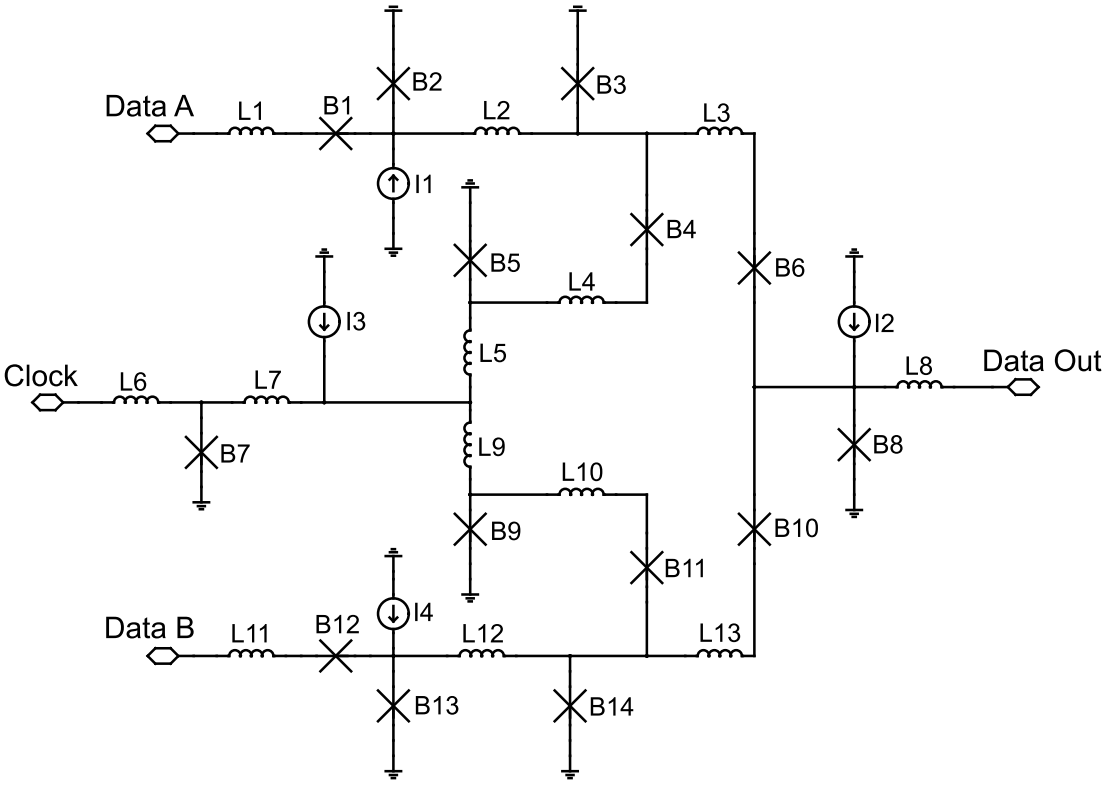


Figure 6.6: RSFQ-AND circuit schematic.

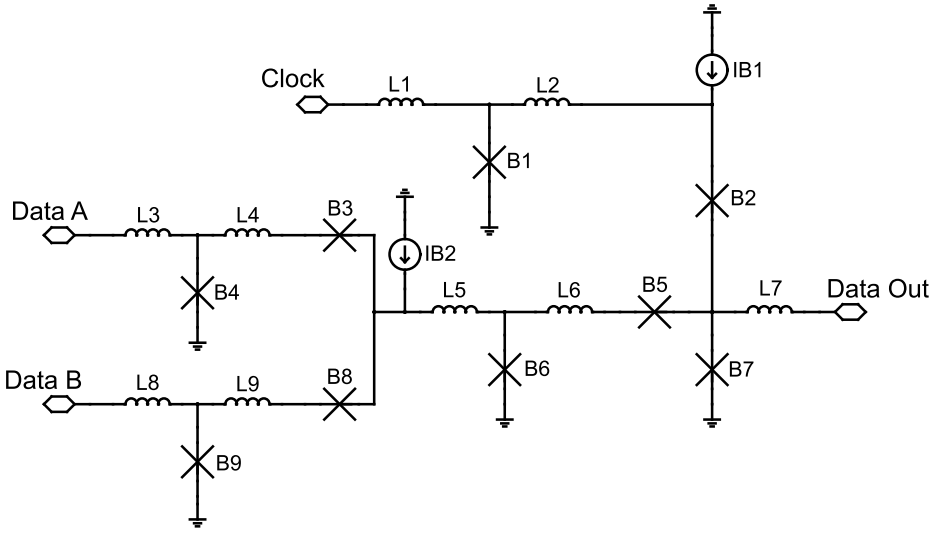


Figure 6.7: RSFQ-OR circuit schematic.

Junctions	Inductors	Bias Currents
$B_1 : 200\,\mu m^2$	$L_1 : 2.47\,pH$	$I_1 : 187\,\mu A$
$B_2 : 250\,\mu m^2$	$L_2 : 9.4p\,H$	$I_2 : 256\,\mu A$
$B_3 : 325\,\mu m^2$	$L_3 : 2.75\,pH$	$I_3 : 460\,\mu A$
$B_4 : 275\,\mu m^2$	$L_4 : 3.01\,pH$	$I_4 : 187\,\mu A$
$B_5 : 175\,\mu m^2$	$L_5 : 0.69\,pH$	
$B_6 : 150\,\mu m^2$	$L_6 : 1.25\,pH$	
$B_7 : 300\,\mu m^2$	$L_7 : 0.81\,pH$	
$B_8 : 350\,\mu m^2$	$L_8 : 1.1\,pH$	
$B_9 : 175\,\mu m^2$	$L_9 : 0.69\,pH$	
$B_{10} : 150\,\mu m^2$	$L_{10} : 3.01\,pH$	
$B_{11} : 275\,\mu m^2$	$L_{11} : 2.47\,pH$	
$B_{12} : 200\,\mu m^2$	$L_{12} : 9.4\,pH$	
$B_{13} : 250\,\mu m^2$	$L_{13} : 2.75\,pH$	
$B_{14} : 325\,\mu m^2$		

Table 6.2: Un-optimised RSFQ AND gate parameters.

The OR gate consists of a single fluxon storage loop with components B_6—L_6—B_5—B_7. An input pulse on *DataA* or *DataB* will be stored in this loop. Junctions B_3 and B_8 are used as escape junctions for the data pulses to ensure that an SFQ pulse is not propagated back into the system via the non-receiving input data port. When two consecutive pulses arrive at *DataA* and *DataB* without an interleaved *Clock* pulse, B_5 will switch, thus ensuring that only one pulse is stored in the storage loop. The arrival of a *Clock* pulse will release the stored fluxon through B_7 to generate an output pulse. B_2 serve as escape junction for the incoming *Clock* pulse if no fluxon is present in the storage loop. Parameters for the OR gate can be found in Table 6.3.

Junctions	Inductors	Bias Currents
$B_1 : 200\,\mu m^2$	$L_1 : 1.27\,pH$	$I_1 : 160\,\mu A$
$B_2 : 200\,\mu m^2$	$L_2 : 1.77\,pH$	$I_2 : 520\,\mu A$
$B_3 : 300\,\mu m^2$	$L_3 : 1.27\,pH$	
$B_4 : 325\,\mu m^2$	$L_4 : 0.65\,pH$	
$B_5 : 250\,\mu m^2$	$L_5 : 3.09\,pH$	
$B_6 : 100\,\mu m^2$	$L_6 : 6\,pH$	
$B_7 : 250\,\mu m^2$	$L_7 : 3,47\,pH$	
$B_8 : 300\,\mu m^2$	$L_8 : 1.27\,pH$	
$B_9 : 325\,\mu m^2$	$L_9 : 0.65\,pH$	

Table 6.3: Un-optimised RSFQ OR gate parameters.

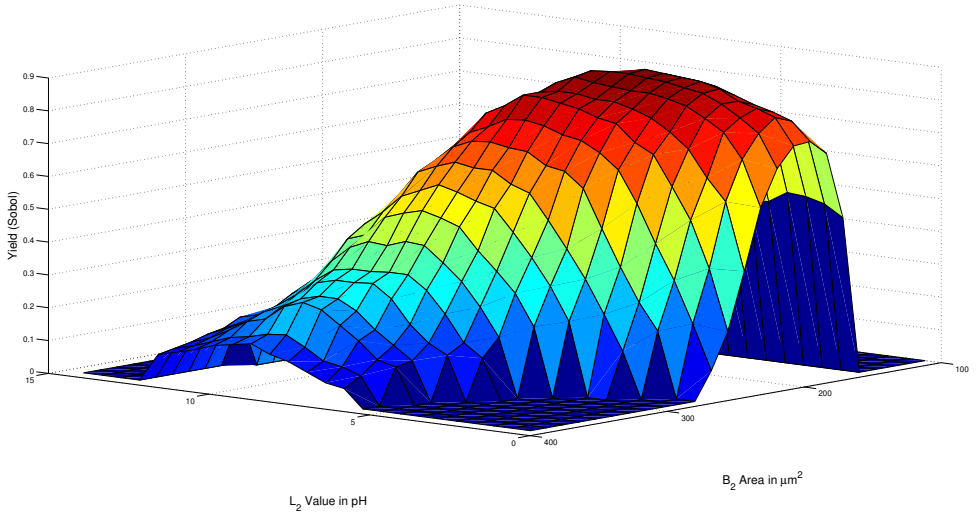


Figure 6.8: Sobol sequence yield map of B_2 and L_2 for the RSFQ AND gate.

6.4 Search space

By looking at the search space of the optimisation problem, one might ascertain which of the algorithms should be superior. The problem faced when optimising circuits, though, is the number of free variables in the system, which make it impossible to visualise the search space as a whole. One can, however, investigate two dimensions of the search space (with an associated merit axis). Consequently, one of the most important features that can be determined is whether the search space is concave (or convex for minimisation problems). This might indicate that a normal greedy heuristic can be used to merely iteratively move through the search space until the global optimum is reached. If the space is not convex, one is more than likely dealing with a multi-modal search space. This is a search space that contains multiple optima which, in general, differ from their optimal values. In such situations more advanced heuristics (such as simulated annealing and genetic algorithms) can assist in finding the global optimum.

Visualisations of the search space were already presented for the critical components in the storage loops in the investigation of artificial neural networks in Chapter 4. These visualisations are again presented below for reference purposes.

From the visualisations it would appear that the yield search space of the two gates are mostly concave, there being only one global maximum. Given the concave structure, the greedy local search algorithm should prove sufficient for the optimisation process. Naturally this is only a two-dimensional (from 19 possible dimensions) view, and the information should be treated as such.

It is also important to bear in mind that only the optimisation of one function, that is the yield value, is considered here. When another merit

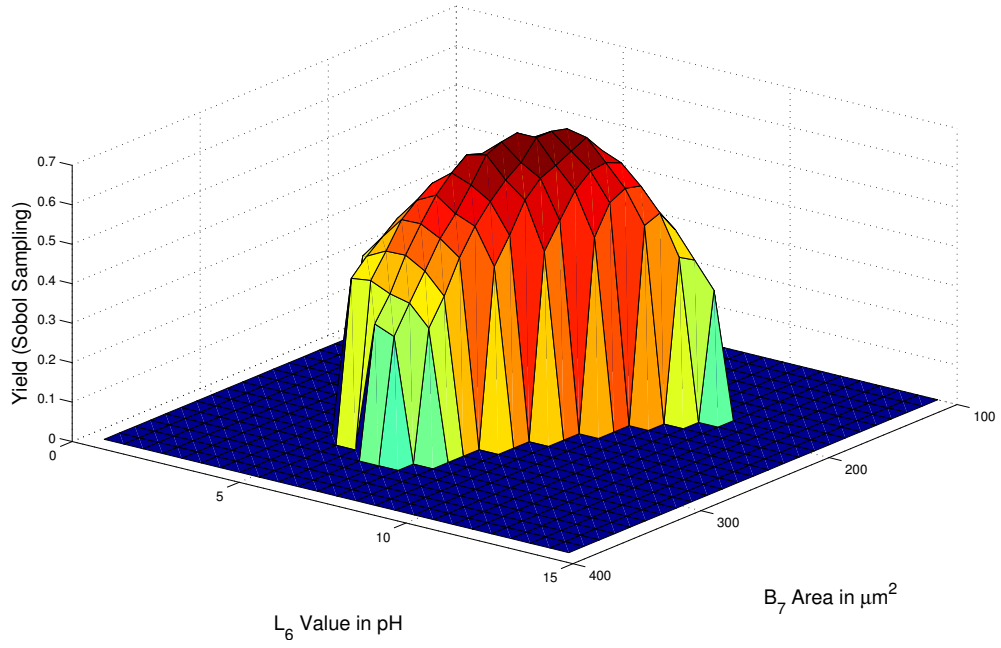


Figure 6.9: Sobol sequence yield map of B_7 and L_6 for the RSFQ OR gate.

function, for example latency, is also considered, a weighted sum is usually employed to garner a final merit value. This could easily result in a multi-modal search space since it is highly likely that the latency of the circuit could have reached an optimum at a position where the yield is not optimal.

6.5 Solution Representation

One of the first steps a designer faces after selecting an optimisation methodology is how to represent the problem to the optimisation algorithm. For the proposed optimisation methods, each component parameter was discretised depending on the upper and lower bounds of the component and the number of steps to be investigated. The bounds and steps are provided by the designer. The step size can then easily be calculated using equation 6.5.1.

$$StepSize = ((UpperBound) - (LowerBound))/Steps \tag{6.5.1}$$

The representation of the components are therefore reduced to a list of integer values. Each integer value corresponds to a component with the range of the integer value being the number of steps for that particular component.

6.6 Testing Methodology

As explained earlier, it is very difficult to compare meta-heuristics due to the randomisation used in the algorithms. An inherently sub-standard algorithm might provide good results simply because of a randomly good starting position. If meta-heuristics are to be compared, it would be crucial to execute a very large number of runs in order to statistically compare the outcomes. This tends to be quite difficult for the optimisation of RSFQ circuits due to the number of SPICE runs needed to sample a single yield point in the search space.

In order to mitigate this situation, it was decided to use the yield space models approximated by artificial neural networks as discussed in Chapter 4. However, as previously illustrated, the approximation of the yield space is not perfect, but the networks do provide a generally good approximation of the overall shape of the search space. Since a comparison of the manner in which the different heuristics move through the search space is of more interest to this study, it is believed that the artificial neural networks can still be a useful tool. Yet, it is still very important to note that the global maximum of the approximation might not be the same as that of the circuit when investigating the search space using the Monte Carlo methods. The artificial neural networks do provide the possibility to execute a very large number of optimisation runs in a fraction of the time that it would have taken the Monte Carlo methods to generate the same data. It is therefore, according to the best knowledge of the author, the only tool available that can statistically compare the methods in a meaningful way and within a practical time frame.

Regarding the networks, it was decided to use Network 5 Run 1 for the AND gate and Network 6 Run 2 for the OR gate as presented in Table 6.4. The reason for this decision was that these networks generally provided good mean squared errors for both training data and test data as well as a good approximation of the two-variable yield maps that were visually investigated.

Gate	Structure	Run	Training MSE	Test MSE
RSFQ-AND	(20,20)	1	0.0020	0.0097
RSFQ-OR	(40,40)	2	0.0072	0.0277

Table 6.4: ANN topologies chosen for optimisation investigation.

It was decided not to make use of networks with three hidden layers even though the mean squared error of their training data tends to be better than that of their two hidden layer counterparts. As pointed out in Chapter 4, networks with more hidden layers tend to be more difficult to optimise given the larger number of free variables in the system. Hence it would be more likely to find a generally good approximation by using two hidden layers even though some details would not be approximated correctly.

Artificial neural networks are easily incorporated as yield approximators for optimisation algorithms. A solution with a structure as depicted in Table 6.4 is normalised between 0 and 1 using the training data limits of the network as normalisation factors. This normalised solution can then be passed on to the network in a feed forward manner which returns a normalised yield value between 0 and 1.

Each parameter step under investigation was run 1000 times in order to identify any statistical trends.

The search space investigated for the optimisation was the same as that used for the artificial neural network in order to ease the input normalisation. Each component therefore had a deviation of plus 150% and minus 150% with absolute bounds to ensure that the circuit was practically implementable. The sample space bounds are listed in Table 6.5 below.

	L	B	I
Abs Min value	$0.5\,pH$	$100\,\mu m^2$	$100\,\mu A$
Abs Max value	$15\,pH$	$400\,\mu m^2$	$600\,\mu A$
Deviation	150 %	150 %	150 %

Table 6.5: Yield optimisation search space bounds.

6.7 Results

6.7.1 Greedy Local Search

As stated earlier, the investigation regarding the greedy local search consists of the neighbourhood size and the neighbourhood dimensions. The parameter points investigated are shown below in Table 6.6.

Number	Neighbourhood Size	Neighbourhood Dimensions
1	1	1
2	1	5
3	1	10
4	5	1
5	10	1
6	5	5
7	10	10

Table 6.6: Greedy local search parameter samples.

Each of the optimisation set-ups were run 1000 times. A stability stopping criteria was used. If no better solution was found for 100 samples, the algorithm

assumed that an optimum had been reached and the algorithm was ended. Each component range defined in Table 6.5 was sub-divided into 100 positions for the generation of the solutions representation.

The results of the greedy local search on the AND gate are shown below. The number of required samples were also recorded in order to ascertain how long each of the 1000 runs took to reach its final stable position. The number of samples that reached a "global" optimum above 90% are also indicated. This should offer a rough indication of how adept an algorithm is at finding the global optimum.

Number	Yield Mean	Yield Max	Yield Std	Mean Samples	Above 90%
1	84.3 %	90.8 %	4.23 %	6887	145
2	84.3 %	90.8 %	4.8 %	4072	153
3	83.9 %	90.7 %	3.8 %	2714	95
4	83.9 %	90.7 %	4.9 %	2699	123
5	83.5 %	90.7 %	4.4 %	1994	91
6	83.5 %	90.7 %	4.1 %	1334	106
7	81.3 %	90.5 %	5.4 %	463	12

Table 6.7: GLS statistical results for the RSFQ AND gate.

It is evident that an increase in neighbourhood size had a significant effect on the number of samples required to obtain an optimum point. This is understandable given the larger "leaps" that solutions are able to take. A slight increase in good solutions (above 90%) were visible using a neighbourhood size of 5. This trend did not continue with a neighbourhood size of 10 though. It could be surmised that the selection caused large leaps to be taken therefore missing small localised areas where the yield is better. This is also revealed in the lower standard deviation of the size 10 neighbourhood which could indicate that localised areas were not sufficiently investigated.

The increase in dimension size had an even larger effect on the number of samples that were needed before a value would settle. It would seem, though, that a global optimum is less likely to be found using this methodology. This makes sense considering that multiple changes to a solution might cause a global point to be missed altogether.

By increasing both the neighbourhood size and the dimensions, the effect on the number of required samples is cumulative. It would seem, though, that the algorithm then loses its ability to find optimum solutions efficiently due to the more random nature of the neighbourhood.

The results of the OR gate are listed in Table 6.8.

The first observation of the results indicate a very large standard deviation at the relatively low mean. This implies that the search space of the OR gate could be substantially different from that of the AND gate. The OR

Number	Yield Mean	Yield Max	Yield Std	Mean Samples	Above 90%
1	56.9 %	95.2 %	40 %	3891	263
2	60.2 %	95.6 %	40 %	2700	376
3	56.3 %	95.7 %	40.8 %	1735	261
4	56.5 %	95.5 %	40.9 %	1699	263
5	57.5 %	95.5 %	40.1 %	1264	231
6	59.8 %	95.5 %	39.8 %	818	251
7	64.3 %	95.3 %	36.8 %	330	186

Table 6.8: GLS statistical results for the RSFQ OR gate.

gate search space possibly has quite a few local optima which tend to lead to sub-optimal solutions being produced.

The results of the OR gate exhibit the same trends as those of the AND gate. Once again, an increase in neighbourhood step size has the effect of decreasing the required samples without a reduction in performance. It would therefore appear that a neighbourhood step size of 5 is a good selection for both the AND gate and the OR gate.

The change in dimensions had the same effect on the samples (reduction) with very little effect on the other metrics. This is slightly in contrast to what was found for the AND gate, which again highlights the difference in the search space of the two gates.

It is interesting to note that to increase both the neighbourhood step and the neighbourhood dimensions (Samples 6 and 7) an overall reduction in the standard deviation of the yield is visible. This could indicate that the local optima in the OR search space are relativy small (geometrically) which allows a more random set-up to jump over these local optima and search further on in the search space.

Generally it would seem that a good option is to use a single dimensional neighbourhood with a step size of 5.

6.7.2 Simulated Annealing

In order to minimise the number of free variables under investigation, the results of the greedy local search were used. The neighbourhood of the simulated annealing algorithm was therefore selected as a single dimensional size with a neighbourhood step of 5.

The free variables of the algorithm comprise of the starting temperature and the temperature steps to zero. As previously explained, the temperature steps to zero multiplied by the iterations per temperature step were selected as a constant. The value of 2000 was selected for this purpose. For example, if a set-up is selected that requests 10 temperature steps before the temperature reaches zero, this would result in $2000/10 = 200$ iterations per temperature step.

Once again 1000 optimisation runs were completed for each sample point, using the same stopping criteria as for the greedy local search tests.

The sample points that were investigated are presented in Table 6.9.

Number	Start Temperature	Temperature Steps
1	0.5	10
2	0.25	10
3	0.125	10
4	0.5	20
5	0.5	40
6	0.25	20
7	0.125	40

Table 6.9: Simulated annealing parameter samples.

The results for the AND gate are shown below in Table 6.10.

Number	Yield Mean	Yield Max	Yield Std	Mean Samples	Above 90%
1	83.8 %	90.7 %	4.1 %	4689	114
2	83.7 %	90.8 %	4.8 %	4743	119
3	83.4 %	90.8 %	4.2 %	4637	88
4	83.9 %	90.8 %	4.2 %	4727	123
5	84.1 %	90.8 %	4.4 %	4761	136
6	83.5 %	90.7 %	4.4 %	4677	99
7	83.9 %	90.8 %	4.4 %	4689	124

Table 6.10: SA statistical results for the RSFQ AND gate.

These results are very similar to that of the greedy local search test. However, there tends to be a slight increase in the mean yield value when more temperature steps are taken, as was the case for samples 5 and 6. One possibility for these similar results is that the search space of the AND gate is very uni-modal which means that it is unlikely for a random greedy search not to find the optimum regularly. Hence, the ability of simulated annealing algorithms to find a global optimum in a multi-modal search space would not be necessary.

The results of the OR gate are shown in Table 6.11.

The results of the OR gate using simulated annealing exhibit substantially better solutions than the greedy local search, in contrast to the AND gate. The mean in general tends to be higher than that of the greedy local search, showing the possibility that the simulated annealing algorithm tends to find optima more regularly in this search space. The most significant improvement, however, was demonstrated when the starting temperature was decreased. This

Number	Yield Mean	Yield Max	Yield Std	Mean Samples	Above 90%
1	63.5 %	95.5 %	38.7 %	3677	312
2	66.2 %	95.4 %	37.4 %	3630	329
3	70.5 %	95.4 %	34.9 %	3559	355
4	63.6 %	95.6 %	38.4 %	3668	298
5	64.6 %	95.4 %	37.7 %	3606	295
6	66.1 %	95.6 %	37.5 %	3596	327
7	69.7 %	95.6 %	35.4 %	3562	349

Table 6.11: SA statistical results for the RSFQ OR gate.

had the effect of reducing the chance that a non-improving solution was accepted prematurely in the optimisation run.

The increase in temperature steps also seemed to add a slight improvement to the algorithm as again demonstrated in samples 5 and 6, but the effect is not as significant as the reduction in the starting temperature.

It was suggested that the search space of the OR gate might be more multimodal than that of the AND gate. This could very well be why the simulated annealing algorithm performs substantially better on the OR gate given the ability of this algorithm to step out of the local optima.

6.7.3 Genetic Algorithm

The variables under test for the genetic algorithm is the generation size, the number of crossover points and the mutation ratio. The mutation operation was chosen to mimic the same neighbourhood search as the previous two algorithms for the sake of consistency. Therefore when a mutation operation occurs, one dimension is varied for the first five steps. The same stopping criteria were applied for the algorithm. If no better solution was found after 100 steps (generations in this case), the algorithm was declared to be complete. The truncation ratio was fixed at 0.5 for all the sample set-ups and the crossover ratio was selected as 0.7.

The samples set-ups of the algorithm are listed in Table 6.12.

Number	Generation Size	Crossover Points	Mutation Ratio
1	40	4	0.25
2	40	4	0.5
3	40	4	0.75
4	40	2	0.5
5	40	8	0.5
6	20	4	0.5
7	80	4	0.5

Table 6.12: Genetic algorithm parameter samples.

As is evident from Table 6.12, a nominal sample set-up with a generation size of 40, 4 crossover points and a mutation ratio of 0.5 was selected. Each variable was then varied one step up and down from this nominal.

The results for the AND gate are shown in Table 6.14.

Number	Yield Mean	Yield Max	Yield Std	Generations	Above 90%
1	86.1 %	90.8 %	3.3 %	867	191
2	86.2 %	90.8 %	3.3 %	718	175
3	86.6 %	90.8 %	3.3 %	1082	266
4	86 %	90.8 %	3.4 %	767	188
5	86.3 %	90.8 %	3.3 %	710	203
6	84.9 %	90.8 %	3.9 %	1008	129
7	87.6 %	90.8 %	2.6 %	573	330

Table 6.13: GA statistical results for the RSFQ AND gate.

In general, the genetic algorithm was able to reach better results than both the greedy local search and simulate annealing algorithms for the AND gate. This is evident in the higher mean and lower standard deviation values. Increasing the mutation ratio above the 0.5 tended to produce slightly better results as can be seen from the number of runs above 90% in sample 3. This could be due to the ability of the local searches implemented by means of mutation to find local optimum points. Reducing the crossover points to 2 from the nominal 4 had the effect of a slight decrease in performance as is evident from the lower mean and higher standard deviation values. It is also interesting to note that the same maximum value was found in each sample, indicating that the algorithm is very adept at finding the global optimum.

The most significant improvement, though, is seen with the increase in the genomes per generation, which also exhibited an increase in the yield mean and a decrease in deviation. It would seem that the extra diversity supplied by more genomes does tend to produce better results.

The number of generations undertaken before a stable value was reached tended to be very high in general. In order to roughly compare the generation count to the sample counts of the greedy local search and the simulated annealing algorithms one has to investigate the number of yield points sampled. For each generation this would be in the order of $Genomes\_Per\_Gen \times Crossover\_Ratio$ since the genomes that are not part of the crossover operation are taken from the current generation and therefore need not have their fitness calculated. As an example, equivalent yield samples for sample 1 are in the range $867 \times 40 \times 0.7 = 24276$ which is substantially more than that of the other two algorithms. In order to mitigate this large range, it is suggested that a yield threshold should rather be used as a stopping criteria. After the yield threshold is reached a pure local search can be implemented to find the local optima of each genome.

Number	Yield Mean	Yield Max	Yield Std	Generations	Above 90%
1	91 %	95.6 %	2.3 %	673	700
2	91.1 %	95.6 %	2.3 %	578	720
3	91.6 %	95.6 %	2 %	866	800
4	90 %	95.6 %	2.2 %	605	709
5	91.2 %	95.6 %	2.2 %	568	741
6	90.4 %	95.6 %	2.6 %	780	623
7	91.8 %	95.6 %	1.8 %	446	852

Table 6.14: GA statistical results for the RSFQ OR gate.

The values obtained for the OR gate are also substantially better than those of both of the other algorithms. In is evident that the genetic algorithm is capable of successfully navigating a multi-modal search space (which is suspected to be the case for the OR gate). The change in mutation ratio showed a small improvement by lowering the standard deviation when increased to 0.75. This is in correlation with what was seen for the AND gate. When the number of crossover points were decreased to 2 a slight drop in the mean was visible, providing slightly poorer results. Similar results are evident in the AND gate investigation.

The most significant influence is once again the genomes per generation which caused a substantial decline when reduced to 20 and supplied a better result when increased to 80. The number of generations needed for the algorithm to run were still high but the large increase in the mean implies that it is likely that the genetic algorithm only has to be run once to obtain favourable results while for the other two algorithms more runs may be necessary.

6.8 Comparison

What is evident from the tests is that the algorithms can have different effects on different search spaces. For the AND gate a normal greedy local search seems to suffice while for the OR gate both the simulated annealing algorithm and the genetic algorithm tend to perform substantially better. As was suggested, this might indicate that the search space of the OR gate is multi-modal given the large standard deviation in its the local search techniques.

It is an open question whether the search space of the OR gate truly is multi-modal or whether the artificial neural network implementation caused the multi-modality. The answer, though, is not crucial to the relevance of these results. What is demonstrated here is that it is highly likely that simulated annealing and genetic algorithms will perform better for search spaces in which normal local searches exhibit a large standard deviation (therefore possibly being multi-modal). Given the complexity of RSFQ circuits, the likelihood of multi-modal search spaces is high. If multi-objective metric values (such as

yield and latency) are added to the algorithms, multi-modality becomes even more likely.

In general, when optimising an RSFQ circuit for yield, it is suggested to commence with a greedy local search or simulated annealing algorithm if available. If after a few runs the standard deviation seems to be very high, a strong indication of a multi-modal search space have been found. The usage of optimisation algorithms designed to circumvent local-optima, such as simulated annealing or genetic algorithms, are then suggested.

The possibility of using a greedy local search to identify a possible multi-modal search space for RSFQ optimisation should be further investigated in future work. The usage of meta-heuristics such as genetic algorithms generally add a substantial time to the computation of an optimal solution and should therefore be used only when necessary. It is suggested that more complex circuitry are also optimised using the methods introduced in this chapter to further ascertain the usability of these algorithms.

Chapter 7

Conclusion and Future Work

Various design automation and optimisation techniques were introduced in this work which have been shown to be beneficial for the development of RSFQ circuits.

In Chapter 3 it was demonstrated that Latin Hypercube Sampling and Sobol sequences can successfully be utilised as variance reduction techniques for yield estimation. A substantial amount of research can still to be undertaken to gain the full benefit of statistical analysis of the circuit parameters on yield. A next step would be to include a principal component analysis (PCA) of the circuit parameters in terms of yield. This analysis technique investigates the correlation between components while also identifying which components are responsible for the most variance. These principal components can then be used for yield approximation, thereby reducing the dimensionality of the sample space. The semiconductor and financial industries also provide many other examples of variance reduction techniques which could be beneficial when used with RSFQ circuits.

In Chapter 4 the possibility of yield estimation by means of artificial neural networks was investigated. It was found that these networks can be used to model the shape of the yield sample space. There is, however, substantial room for improvement since it was determined that the artificial neural network approximation does not perfectly fit the Sobol based sample space used for the comparison. Various choices in network structure, activation function, parameter settings and training algorithms should be further investigated to ascertain the optimum topology for RSFQ yield approximations. Artificial neural networks constitute an extensive field of research and provide many other avenues of exploration for yield estimation. Artificial neural networks can also be directly used for optimisation, as can be seen with the Hopfield networks investigated in [58]. Currently though, the amount of training data needed might dissuade users from using normal multi-level Perceptron networks for direct yield approximation. The usage of such a network as a tool for comparing the efficacy of optimisation algorithms was, however, demonstrated to be of great use as was described in Chapter 6. Stochastic optimi-

sation algorithms generally have many parameters that affect the outcome of the optimisation run. Using an artificial neural network approximation allows optimisation runs to complete in a fraction of the time needed by Monte Carlo approximations. This provides a statistical method to compare algorithms and algorithm variants.

In Chapter 5 an automated state machine extraction algorithm was introduced that can be used for functional circuit testing as well as for the creation of HDL models. In terms of functional testing, the algorithm was shown to be successful and was the primary method used to prove state machine equivalence to nominal circuits for all optimisation and Monte Carlo methods used in this work. Various improvements can still be done to further refine the method. In terms of automation, improvements could include the automatic identification of fluxon storage loops (without an exhaustive search). Theoretical justification can also be investigated for the constants used in the method that currently only have an empirical foundation.

Two approaches to critical timing extraction was also presented in Chapter 5, one which makes use of empirically identifying the point at which the circuit fails and another based on current stability. Further work can be used to validate which critical timing model match practical circuit failures more closely. A further study should also include a Monte Carlo of the critical timing parameters to ascertain how these values change under process variations. Statistical timing information could also be incorporated in future implementations as was proposed in [59].

The HDL models created should also be practically tested when combined to form larger circuits. This could identify further work in leakage current modeling which could also be included in the HDL models.

In Chapter 6 various optimisation algorithms were compared for their applicability for RFSQ circuits. A novel technique was used to compare the algorithms using an artificial neural network approximation of the yield sample space as elaborated upon in Chapter 4. Given the considerable difference in optimisation results one can surmise that the yield sample spaces of the two gates investigated are substantially different. Both genetic algorithms and simulated annealing are known to perform better when dealing with multi-modal problems, therefore one can surmise that the OR gate sample space contains many local optima. The possibility was pointed out that the variance of results obtained using a greedy algorithm can be used as an indication of a multi-modal search space. This could be due to the search algorithm failing to leave a local optimum and thus finding substantially different results depending on the algorithm's starting position. When a large variance in optimisation results is obtained it would be beneficial to make use of algorithms which are more apt at traversing a multi-modal search space. By using the artificial neural network approximation the effects of algorithm parameter variation was also successfully compared. It is clear that certain parameters, such as generation size for genetic algorithms, have the largest influence on the optimisation

results. This method can therefore be used in future to ascertain the influence of algorithmic parameters.

Future work in optimisation can also include further refinements to artificial neural network models for algorithm comparison and also the investigation of multi-objective optimisation (latency, leakage current), which have a high possibility of creating multi-modal search spaces.

This work introduced various algorithms and methodologies that further the state of electronic design automation in RSFQ circuits. In terms of yield approximations, automated techniques were proposed to simplify the creation of exhaustive test benches for the use in Monte Carlo runs. Various variance reduction techniques were shown to also reduce the necessary amount of samples needed for a particular confidence interval and in doing so, reduce the time of yield calculations. Artificial neural networks were shown to be valid approximators of the yield sample space which is of significant use in statistically quantifying the performance of different optimisation algorithms. Further work is, however, necessary before the Monte Carlo methods can be completely replaced as the yield approximators used in everyday RSFQ circuit design.

There is a high possibility that HDL models will play a crucial role in the creation of complex RSFQ circuits in the same manner that these models form a critical part in the design cycle of semiconductor circuits. This work proposed an automated method of extracting HDL models in an exhaustive and repeatable manner for the creation of high-level cell libraries. With further refinements in leakage current, these models can form the core of complex RSFQ circuitry.

All the methods presented in this work forms part of a software library that can be integrated with existing development environments or form a basis for future work.

Appendices

Appendix A

RSFQ Automation and Optimisation Library

A.1 Introduction

The functionality described in this work forms part of a extensible software library ,written in C++, which can be used for general RSFQ circuit investigation as well as optimisation (mulit-objective or otherwise). Currently the functionality provided is:

- Automated extraction of the state machine representation of an RSFQ circuit provided in SPICE format.
- Automated generation of a VHDL model by using the extracted state machine representation and a timing extracted technique (either empiric or stability).
- Yield estimation of circuit for which a state machine has been extracted. The estimation can be using crude Monte Carlo, Latin Hypercube Sampling or a Sobol sequence.
- The creation of an artificial neural network to approximate single or multi-modal objective functions.
- The optimisation (in terms of yield, latency or both) of an RSFQ circuit provided in SPICE format.
- Various miscellaneous abilities useful for investigating a circuit's search space.

This chapter will explain classes, their interconnections and their methods in order to allow future work to be done on the software library. Note that the library has been designed to function with a Linux based operating system.

There are however only a few operating specific sections (such as the multi-core processing etc.) which should portable to other operating systems. Note also that their does not currently exist a graphical user interface to facilitate easy use of the library. Currently all commands are passed via text files. It should be relatively easy to create a graphical front-end to improve usability.

The dependencies are as follow:

- Boost C++ library 1.56 or higher. http://www.boost.org
- JSIM or JSPICE available from various resources
- Sobol initial values for yield calculation. http://web.maths.unsw.edu.au/ fkuo/-sobol/

The project was created using the free CodeLite IDE under Ubuntu 14.04.

In the next section, an overview of the classes and their relationships will be provided. A brief description of each method will also be provided. After the descriptions a detailed discussion will explain the usage of the program via the text files and the options available.

Keep in mind that this library is under active development, and will more than likely be restructured in a more organised way in the near future.

A.2 Library Overview

In this section each class will be investigated with it's interfaces to other classes. Note that a solid line in the figures indicate ownership. The class object being pointed to was therefore created by the base class and will be responsible for its cleanup. A dotted line means that a class makes use of a certain object but does not own it. Please note that these are not UML diagrams. Since the library is under active development, only the classes as a whole will be discussed.

A.2.1 App

The App class is the main entry point for the library. It contains the methods for invoking all the functionality available to the library. The App class is controlled via a configuration text file created by the user. This will be explained later.

The App will always create a SPICE_File objects from the supplied SPICE file and will also automatically try to extract the state machine representation. The state machine representation is necessary for all the other classes to function correctly. Once the SPICE_File object is created the user's required operation is executed.

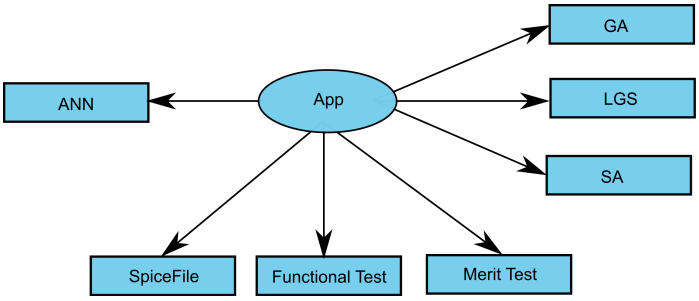


Figure A.1: App Class

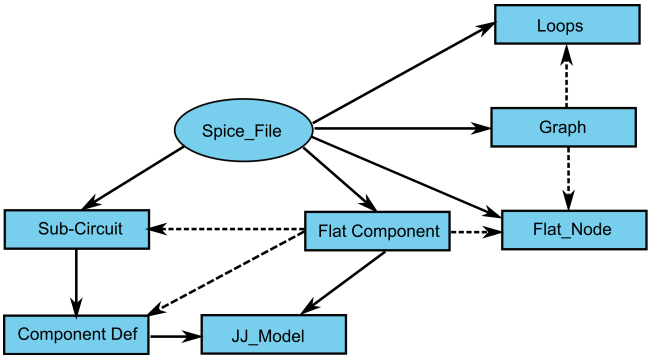


Figure A.2: SPICE File Class

A.2.2 SPICE File

The SPICE_File class is used to parse the spice file and create a flat hierarchy for the extraction of possible inductive loops.

The parsing process creates sub-circuit objects containing the component definitions of each component found. For each Josephson junction found an additional junction model is also created and associated with the relevant component definition In order to extract the state machine representation the SPICE hierarchy needs to be flat. For each instance of the sub-circuits under test a unique flat component if create that uses unique node numbers. For each junction flat component a unique junction model is also created. With the flat hierarchy created, a graph is created using the flat nodes as nodes and the flat components as vertexes. The graph is created using the Boost C++ Graph library. After the graph's creation, all the possible loops in the circuit is identified and stored using the algorithm described in Chapter 5.

A.2.3 Functional Test

The functional test class is used to extract the state machine representation of the parsed SPICE file. This is done using the methodology explained in Chapter 5. The state transitions, outputs generated and state identifiers are all stored in the State objects. After the initial state machine extraction, it

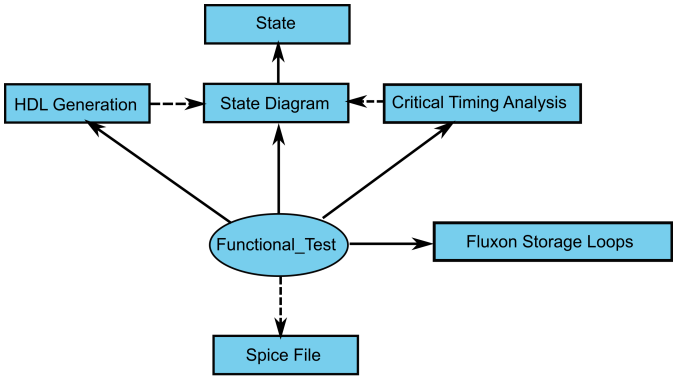


Figure A.3: Functional Test Class

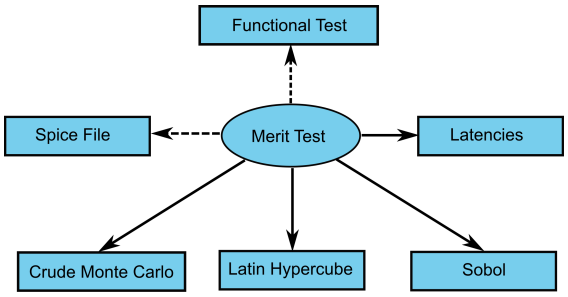


Figure A.4: Merit Test Class

is known which of the loops identified by the SPICE File opbject are actually used for fluxon storage. These values are then saved in the Functional Test object for use with merit testing. After the State Machine extraction process a critical timing analysis can be done on the circuit as described in Chapter 5 after which an HDL model can be generated.

A.2.4 Merit Test

The merit test class is used to calculate merit values for yield analysis and also optimisation purposes. The circuit yield is calculated via either crude Monte Carlo, Latin Hypercube sampling or Sobol sequences. The supplied SPICE file is used as the source for the yield estimation and might contain component values different to the nominal SPICE file when calculating the yield for optimisation algorithms. The option to return all the latencies of the circuit also exists. The merit function itself does not assign a weighting value to the multiple criteria if used.

A.2.5 ANN

The ANN class governs the creation of an artificial neural network. This normally necessitates the generation of training data. Using the SPICE File

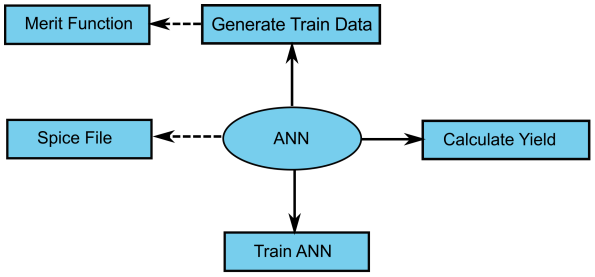


Figure A.5: Merit Test Class

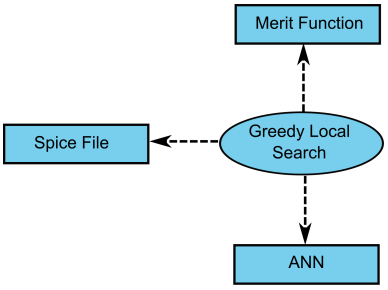


Figure A.6: Greedy Local Search Class

various sampling techniques can be used to test the sample space of the circuit. The merit value of the sample point can then be found using the merit function. After training data has been created, the artificial neural network can be trained via user defined parameters to be described below.

The trained ANN can then be used by optimisation algorithms to approximate the yield value without calling the merit function.

A.2.6 Greedy Local Search

The greedy local search class implements a local search algorithm as explained in Chapter 6. The SPICE File to optimise is supplied to the algorithm which makes a copy of the spice object. The merit values can be calculated using Monte Carlo Methods or a artificial neural network.

A.2.7 Simulated Annealing Algorithm

The Simulated annealing class implements the simulated annealing optimisation algorithm. As was done for the LGS algorithm, a SPICE File is supplied which is used as source for the optimisation. Once again merit values can be obtained via Monte Carlo methods or from a trained ANN.

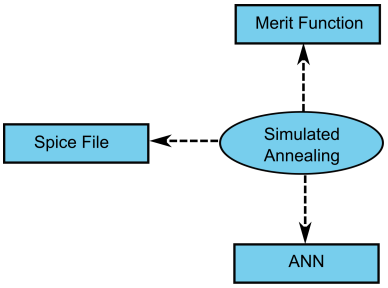


Figure A.7: Simulated Annealing Class

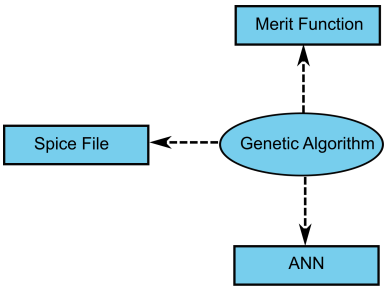


Figure A.8: Merit Test Class

A.2.8 Genetic Algorithm

The GA class implements the Genetic Algorithm optimisation algorithm as explained in Chapter 6. The SPICE File which needs to be optimised is provided to the GA of which a copy is made in order to vary values at will. The merit value of the SPICE File under test can be gathered using the Monte Carlo based merit function or the trained artificial neural network.

A.3 Basic User Guide

The program functions by passing a configuration file containing instructions, other configuration files and parameters as a command line argument. The various configuration files will be discussed in this section. All the configuration files makes use of a parameter keyword on a new line followed by space separated values. The full user guide will be available on [60] as the library reaches as stable release.

A.3.1 Main Configuration File

Tables A.1 and A.2 contains the commands and parameters that can be set using the main configuration file.

Parameter	Options	Comment
COMMAND	EXTRACT_STATE_MACHINE	Extract the state machine of the supplied SPICE file
	GENERATE_HDL	Generate a VHDL representation of the supplied SPICE file
	YIELD_ANALYSIS	Do a yield analysis on the supplied SPICE file
	RUN_ANN	Create/Train a Artificial Neural Network
	RUN_GLS	Run Greedy Local Search on supplied SPICE file
	RUN_SA	Run the Simulated Annealing Algorithm on supplied SPICE file
	RUN_GA	Run Genetic Algorithm on supplied SPICE file
SPICE_TYPE	JSIM	Use JSIM as simulator
	JSPICE	Use JSPICE as simulator
FUNC_CONFIG	(path to file)	Path to the file containing information for state machine extraction
TOLERANCE_FILE	(path to file)	Path to the file containing the process variations for Monte Carlo
JUNCTION_INFO_FILE	(path to file)	Path to the file containing junction area, capacitance and shunt resistance values
ANN_CONFIG_FILE	(path to file)	Path to file containing information used by the ANN functionality

Table A.1: Main Configuration File Parameters

OPT_CONFIG_FILE	(path to file)	Path to file containing optimisation information
OUTPUT_FILE	(path to file)	Path to the output file that should contain the relevant results
MONTE_TYPE	CMC	Select crude Monte Carlo as yield estimator
	LHS	Select Latin Hypercube Sampling as yield estimator
	SOBOL	Select Sobol sequence as yield estimator
MONTE_RUNS	(unsigned integer number)	Number of Monte Carlo samples for yield estimation
LHS_SAMPLES	(unsigned integer number)	Number of sample points for LHS yield estimation
SOBOL_POINTS	(unsigned integer number)	Number of points used by Sobol sequence for yield estimation
SOBOL_RANDOM	(1 or 0)	If one randomise the Sobol sequence by shuffling the entries
SOBOL_INIT_FILE	(path to file)	Path to file containing Sobol initialisation values
SPICE_FILE	(path to file)	Path to the SPICE file under investigation
COMP_LINK_FILE	(path to file)	Path to file indicating component links
AUTO_ADD_RS	(1 or 0)	If 1 automatically calculate junction shunt resistances for a $\mathcal{B}_c$ of 1
IGNORE_SUB_CIRS	(list of sub-circuit names)	List of sub-circuits in the SPICE file that should be ignored

Table A.2: Main Configuration File Parameter Cont.

A.3.2 Functional Configuration File

Table A.3 contains the available set-up parameters used by the state machine extraction algorithm.

INPUTS	(integer list)	Space separated list of nodes that serve as inputs to the circuit
INPUTS_JJ	(string list)	Space separated list of full junction names that is associated with the input above in INPUTS
OUTPUTS	(string list)	Space separated list of full junction names that serve as outputs to the circuit
TIME_OFFSET	(intege value)	The time between pulses for the extraction of the state machine representation
RULE	(list of rule string)	Applies a rule for the extraction process. See Chapter 5 for details

Table A.3: Functional Configuration File Parameter Cont.

A.3.3 Greedy Local Search Configuration File

Table A.4 and A.5 contains the optimisation parameters available when running the greedy local search algorithm.

OUTPUTS	(string list)	Space separated list of full junction names that are the outputs of the circuit
TIME_OFFSET	(integer)	An integer value representing the number of pico seconds before new stimulus can be applied to the circuit
COMPONENTS	(list of component names)	SPICE components to be optimised
	ALL_B | ALL_L | ALL_I	Used to indicate that all of a type of component should be optimised
SAMPLE_TYPE	RELATIVE | ABSOLUTE	Create the search space relative to the current component values or use the absolute values supplied
MIN_VALUES	(list of component values)	The absolute minimum value of the search space associated with the COMPONENTS listed

Table A.4: Greedy Local Search Parameters

MAX_VALUES	(list of component values)	The absolute maximum value of the search space associated with the COMPONENTS listed
COMP_DEV	(list of deviation values)	The deviations (+ and -) that should be applied to create the search space if RELATIVE is chosen
B_STEPS	(unsigned integer)	The number of steps in the search space for the junction range
L_STEPS	(unsigned integer)	The number of steps in the search space for the inductor range
I_STEPS	(unsigned integer)	The number of steps in the search space for the bias current range
NEIGHBORHOOD_DIMS	(unsigned integer)	The number of dimensions used to create a neighbor
NEIGHBORHOOD_SIZE	(unsigned integer)	The maximum number of steps used to create a neighbor (randomised)
MONTE_STOP_VALUE	(value between 0 and 1)	Stop the algorithm when this normalised yield value is reached
SAMPLES_STOP	(unsigned integer)	Stop the algorithm when this number of steps were taken
MERIT_METHOD	ANN | MONTE	Use the current ANN for merit values or use the set-up Monte Carlo method

Table A.5: Greedy Local Search Paramaters Cont.

A.3.4 Simulated Annealing Configuration File

Table A.6 and A.7 contains the optimisation parameters available when running the simulated annealing algorithm.

COMPONENTS	(list of component names)	SPICE components to be optimised
	ALL_B | ALL_L | ALL_I	Used to indicate that all of a type of component should be optimised
SAMPLE_TYPE	RELATIVE | ABSOLUTE	Create the search space relative to the current component values or use the absolute values supplied
MIN_VALUES	(list of component values)	The absolute minimum value of the search space associated with the COMPONENTS listed
MAX_VALUES	(list of component values)	The absolute maximum value of the search space associated with the COMPONENTS listed
COMP_DEV	(list of deviation values)	The deviations (+ and -) that should be applied to create the search space if RELATIVE is chosen
B_STEPS	(integer value)	The number of steps in the search space for the junction range
L_STEPS	(integer value)	The number of steps in the search space for the inductor range
I_STEPS	(integer value)	The number of steps in the search space for the bias current range

Table A.6: Simulated Annealing Parameters

NEIGHBORHOOD_DIMS	(integer value)	The number of dimensions used to create a neighbor
NEIGHBORHOOD_SIZE	(integer value)	The maximum number of steps used to create a neighbor (randomised)
MONTE_STOP_VALUE	(value between 0 and 1)	Stop the algorithm when this normalised yield value is reached
SAMPLES_STOP	(integer value)	Stop the algorithm when this number of steps were taken
MERIT_METHOD	ANN | MONTE	Use the current ANN for merit values or use the setup for the Monte Carlo method
START_TEMP	(float value)	Specifies the starting temperature for the algorithm
ITER_PER_TEMP	(integer value)	Specifies to the number of iterations per temperature step
TEMP_STEP	(float value)	Specifies the temperature step for the algorithm

Table A.7: Simulate Annealing Parameters Cont.

A.3.5 Genetic Algorithm Configuration File

Table A.8 and A.9 contains the optimisation parameters available when running the genetic algorithm.

COMPONENTS	(list of component names)	SPICE components to be optimised
	ALL_B | ALL_L | ALL_I	Used to indicate that all of a type of component should be optimised
SAMPLE_TYPE	RELATIVE | ABSOLUTE	Create the search space relative to the current component values or use the absolute values supplied
MIN_VALUES	(list of component values)	The absolute minimum value of the search space associated with the COMPONENTS listed
MAX_VALUES	(list of component values)	The absolute maximum value of the search space associated with the COMPONENTS listed
COMP_DEV	(list of deviation values)	The deviations (+ and -) that should be applied to create the search space if RELATIVE is chosen
B_STEPS	(integer value)	The number of steps in the search space for the junction range
L_STEPS	(integer value)	The number of steps in the search space for the inductor range
I_STEPS	(integer value)	The number of steps in the search space for the bias current range

Table A.8: Genetic Algorithm Parameters

NEIGHBORHOOD_DIMS	(integer value)	The number of dimensions used to mutate a genome
NEIGHBORHOOD_SIZE	(integer value)	The maximum number of steps used to mutate a genome (randomised)
MONTE_STOP_VALUE	(value between 0 and 1)	Stop the algorithm when this normalised yield value is reached
GENERATIONS_STOP	(integer value)	Stop the algorithm when this number of generations have been created
MERIT_METHOD	ANN | MONTE	Use the current ANN for merit values or use the setup for the Monte Carlo method
GENOMES_PER_GEN	(integer value)	Number of genomes that should be created per generation
TRUNC_RATIO	(float value between 0 and 1)	The ratio of the current generation that forms part of the gene pool
MUTATE_RATIO	(float value between 0 and 1)	The probability of mutating a specific genome
CROSSOVER_POINTS	(integer value)	The number of points used in the crossover operation
CROSSOVER_RATIO	(float value between 0 and 1)	The ratio of a new generation that should be created using crossover

Table A.9: Genetic Algorithm Parameters Cont.

A.3.6 Artificial Neural Network Configuration File

Table A.10 and A.11 contains the parameters available when creating a artificial neural network.

COMPONENTS	(list of component names)	SPICE components to be optimised
	ALL_B | ALL_L | ALL_I	Used to indicate that all of a type of component should be optimised
SAMPLE_TYPE	RELATIVE | ABSOLUTE	Create the search space relative to the current component values or use the absolute values supplied
MIN_VALUES	(list of component values)	The absolute minimum value of the search space associated with the COMPONENTS listed
MAX_VALUES	(list of component values)	The absolute maximum value of the search space associated with the COMPONENTS listed
COMP_DEV	(list of deviation values)	The deviations (+ and -) that should be applied to create the search space if RELATIVE is chosen
NETWORK_STRUCTURE	(list of integer)	Indicate the hidden layers of the ANN
TRAIN_ANN	(0 or 1)	1 indicates the created, or loaded ANN should be trained given training data
TRAIN_DATA_METHOD	CMS | LHS | LOCAL_LHS	Choose the method that will be used to generate training data
TRAIN_DATA_STEPS	(integer value)	The number of training data samples to generate
SAVE_TRAIN_DATA	(0 or 1)	1 indicates that the training data should be saved for later use

Table A.10: Artificial Neural Network Parameters

READ_TRAIN_DATA	(0 or 1)	1 indicates that the training data should be read from the file specified
TRAIN_DATA_FILENAME	(path to file)	The file where the training data should be saved or restored from
TRAIN_TEST_DATA_FILENAME	(0 or 1)	The file where the test data is located for training purposes
READ_ANN	(0 or 1)	1 indicates that the ANN should be read from the file specified
SAVE_ANN	(0 or 1)	1 indicates that the created ANN should be saved to the file specified
ANN_SAVE_FILENAME	(path to file)	File where ANN should be saved or read from

Table A.11: Artificial Neural Network Parameters Cont.

A.3.7 Configuration File Examples

Examples of the configuration files are shown below for an RSFQ AND gate.

```
SPICE_FILE rsfq_and.cir
SPICE_TYPE jsim
FUNC_CONFIG func_and.txt
TOLERANCE_FILE tol_and.txt
JUNCTION_FILE junctions.txt
ANN_CONFIG_FILE ann_config_and.txt
MONTE_RUNS 1000
MONTE_TYPE cms
IGNORE_SUB_CIRCUITS XDCSFQ XTKL250
AUTO_ADD_RS 1
COMP_LINK_FILE comp_links_and.txt
```

Figure A.9: Main Configuration Example

```
L1_X5 L11_X5
B1_X5 B12_X5
B2_X5 B13_X5
I1_X5 I4_X5
L2_X5 L12_X5
B3_X5 B14_X5
L3_X5 L13_X5
B4_X5 B11_X5
L4_X5 L10_X5
B5_X5 B9_X5
L5_X5 L9_X5
B6_X5 B10_X5
```

Figure A.10: Linked Components Example

```
INPUTS 1 6 10
INPUTS_JJ X3_B2 X7_B2 X1_B2
OUTPUTS X6_B1
TIME_OFFSET 100
```

Figure A.11: Functional Configuration Example

```
L_LOCAL 0.13
L_GLOBAL 0.13
B_GLOBAL 0.13
B_LOCAL 0.13
I_LOCAL 0.13
I_GLOBAL 0.13
```

Figure A.12: Tolerance Configuration Example

```
100u 0.498988p 2.576
125u 0.624488p 2.057
150u 0.785522p 1.695
175u 0.883018p 1.456
200u 1.00852p 1.274
225u 1.13151p 1.135
250u 1.26153p 1.022
275u 1.38502p 0.9278
300u 1.50550p 0.8529
325u 1.62999p 0.7879
350u 1.75851p 0.7308
375u 1.87848p 0.6842
400u 2.01704p 0.6371
425u 2.15234p 0.5934
450u 2.28123p 0.5435
```

Figure A.13: Junction Configuration Example

```
MUTATE_RATIO 0.5
CROSSOVER_RATIO 0.7
CROSSOVER_POINTS 3
MUTATION_POINTS 1
GENERATIONS 10
TRUNC_RATIO 0.5
COMPONENTS ALL_L ALL_B ALL_I
MIN_VALUES 0.5p 100u 100u
MAX_VALUES 15p 400u 600u
COMP_DEV 1.5 1.5 1.5
NEIGHBORHOOD_SIZE 5
SAMPLE_TYPE RELATIVE
B_STEPS 100
L_STEPS 100
I_STEPS 100
MONTE_STOP_VALUE 0.95
GENERATIONS_STOP 100
MERIT_METHOD ANN
GENOMES_PER_GEN 20
```

Figure A.14: Genetic Algorithm configuration Example

```
COMPONENTS ALL_L ALL_B ALL_I
MIN_VALUES 0.5p 100u 100u
MAX_VALUES 15p 400u 600u
SAMPLE_TYPE relative
COMP_DEV 1.5 1.5 1.5
EPOCHS 6000
NETWORK_STRUCTURE 20 20
TRAIN_DATA_METHOD local_lhs
TRAIN_DATA_STEPS 4000
TRIAN_ANN 1
SAVE_TRAIN_DATA 0
READ_TRAIN_DATA 1
SAVE_ANN 1
READ_ANN 1
TRAIN_DATA_FILENAME ann_train_and.dat
TRAIN_DATA_TEST_FILENAME ann_train_and_test.dat
ANN_SAVE_FILENAME ann_save.dat
```

Figure A.15: Artificial Neural Network configuration Example

Appendix B

Results of ANN Training Runs

This chapter contains the results of the ANN training procedure for the RSFQ AND and OR gates.

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10)	1	0.0039	0.0125
(10)	2	0.0040	0.0131
(10)	3	0.0044	0.0113
(10)	4	0.0041	0.0024
(10)	5	0.0040	0.0117

Table B.1: ANN Results for AND gate Structure 1

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20)	1	0.0023	0.0090
(20)	2	0.0027	0.0099
(20)	3	0.0026	0.0129
(20)	4	0.0027	0.0111
(20)	5	0.0027	0.0117

Table B.2: ANN Results for AND gate Structure 2

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40)	1	0.0020	0.0120
(40)	2	0.0020	0.0091
(40)	3	0.0024	0.0113
(40)	4	0.0021	0.0115
(40)	5	0.0020	0.0087

Table B.3: ANN Results for AND gate Structure 3

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10,10)	1	0.0034	0.0106
(10,10)	2	0.0034	0.0112
(10,10)	3	0.0033	0.0085
(10,10)	4	0.0037	0.0108
(10,10)	5	0.0034	0.0096

Table B.4: ANN Results for AND gate Structure 4

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20,20)	1	0.0020	0.0097
(20,20)	2	0.0017	0.0097
(20,20)	3	0.0019	0.0121
(20,20)	4	0.0020	0.0089
(20,20)	5	0.0017	0.0197

Table B.5: ANN Results for AND gate Structure 5

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40,40)	1	0.0011	0.0119
(40,40)	2	0.0083	0.0110
(40,40)	3	0.0011	0.0097
(40,40)	4	0.0099	0.0106
(40,40)	5	0.0012	0.0113

Table B.6: ANN Results for AND gate Structure 6

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10,10,10)	1	0.0028	0.0133
(10,10,10)	2	0.0035	0.0124
(10,10,10)	3	0.0029	0.0136
(10,10,10)	4	0.0030	0.0121
(10,10,10)	5	0.0032	0.0098

Table B.7: ANN Results for AND gate Structure 7

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20,20,20)	1	0.0013	0.0084
(20,20,20)	2	0.0016	0.0097
(20,20,20)	3	0.0015	0.0060
(20,20,20)	4	0.0014	0.0112
(20,20,20)	5	0.0016	0.0083

Table B.8: ANN Results for AND gate Structure 8

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40,40,40)	1	0.00075	0.0103
(40,40,40)	2	0.00074	0.0120
(40,40,40)	3	0.00076	0.0099
(40,40,40)	4	0.00057	0.0090
(40,40,40)	5	0.00059	0.0107

Table B.9: ANN Results for AND gate Structure 9

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10)	1	0.0211	0.0334
(10)	2	0.0208	0.0310
(10)	3	0.0221	0.0319
(10)	4	0.0225	0.0311
(10)	5	0.0230	0.0346

Table B.10: ANN Results for OR gate Structure 1

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20)	1	0.0177	0.0293
(20)	2	0.0178	0.0272
(20)	3	0.0180	0.0293
(20)	4	0.0204	0.0314
(20)	5	0.0176	0.0289

Table B.11: ANN Results for OR gate Structure 2

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40)	1	0.0163	0.0279
(40)	2	0.0162	0.0296
(40)	3	0.0158	0.0316
(40)	4	0.0167	0.0278
(40)	5	0.0159	0.0282

Table B.12: ANN Results for OR gate Structure 3

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10,10)	1	0.0184	0.0306
(10,10)	2	0.0176	0.0283
(10,10)	3	0.0186	0.0286
(10,10)	4	0.0179	0.0321
(10,10)	5	0.0191	0.0308

Table B.13: ANN Results for OR gate Structure 4

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20,20)	1	0.0115	0.0283
(20,20)	2	0.0116	0.0285
(20,20)	3	0.0115	0.0274
(20,20)	4	0.0118	0.0277
(20,20)	5	0.0120	0.0281

Table B.14: ANN Results for OR gate Structure 5

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40,40)	1	0.0065	0.0295
(40,40)	2	0.0072	0.0277
(40,40)	3	0.0073	0.0285
(40,40)	4	0.0066	0.0279
(40,40)	5	0.0066	0.0258

Table B.15: ANN Results for OR gate Structure 6

Structure	Run	Lowest Train MSE	Lowest Test MSE
(10,10,10)	1	0.0193	0.0324
(10,10,10)	2	0.0200	0.0335
(10,10,10)	3	0.0157	0.0259
(10,10,10)	4	0.0197	0.0311
(10,10,10)	5	0.0182	0.0315

Table B.16: ANN Results for OR gate Structure 7

Structure	Run	Lowest Train MSE	Lowest Test MSE
(20,20,20)	1	0.0114	0.0298
(20,20,20)	2	0.0111	0.0284
(20,20,20)	3	0.0118	0.0294
(20,20,20)	4	0.0112	0.0283
(20,20,20)	5	0.0109	0.0286

Table B.17: ANN Results for OR gate Structure 8

Structure	Run	Lowest Train MSE	Lowest Test MSE
(40,40,40)	1	0.00049	0.0290
(40,40,40)	2	0.00046	0.0286
(40,40,40)	3	0.00046	0.0286
(40,40,40)	4	0.00049	0.0267
(40,40,40)	5	0.00043	0.0276

Table B.18: ANN Results for OR gate Structure 9

Bibliography

[1] K. Likharev and V. Semenov, "Rsfq logic/memory family: a new josephson-junction technology for sub-terahertz-clock-frequency digital systems," *Applied Superconductivity, IEEE Transactions on*, vol. 1, no. 1, pp. 3 –28, mar 1991.

[2] M. Dorojevets, C. Ayala, N. Yoshikawa, and A. Fujimaki, "8-bit asynchronous sparse-tree superconductor rsfq arithmetic-logic unit with a rich set of operations," *Applied Superconductivity, IEEE Transactions on*, vol. 23, no. 3, pp. 1 700 104–1 700 104, June 2013.

[3] V. Michal, E. Baggetta, M. Aurino, S. Bouat, and J.-C. Villegier, "Superconducting rsfq logic: Towards 100ghz digital electronics," in *Radioelektronika (RADIOELEKTRONIKA), 2011 21st International Conference*, April 2011, pp. 1–8.

[4] A. Kadin, *Introduction to Superconducting Circuits*. John Wiley and Sons, 1999.

[5] L. Müller, H. Gerber, and C. Fourie, "Review and comparison of rsfq asynchronous methodologies," *Journal of Physics: Conference Series*, vol. 97, no. 1, 2008.

[6] "Xilinx programmable devices," http://www.xilinx.com/.

[7] "Mathworks technical computing," http://www.mathworks.com/.

[8] C. J. Fourie and M. H. Volkmann, "Status of superconductor electronic circuit design software," *Applied Superconductivity, IEEE Transactions on*, vol. 23, no. 3, june 2013.

[9] T. Van Duzer and C. W. Turner, *Principles of Superconductive Devices and Circuits*. Prentice Hall, 1998.

[10] W. Anacker, "Josephson computer technology," *special issue of IBM J. Res. Devel*, vol. 24, pp. 105–264, March 1980.

[11] N. Fujimaki, S. Kotani, T. Imamura, and S. Hasuo, "Josephson modified variable threshold logic gates for use in ultra-high-speed lsi," *Electron Devices, IEEE Transactions on*, vol. 36, no. 2, pp. 433–446, Feb 1989.

[12] W. Perold, M. Jeffery, Z. Wang, and T. Van Duzer, "Complementary output switching logic-a new superconducting voltage-state logic family," *Applied Superconductivity, IEEE Transactions on*, vol. 6, no. 3, pp. 125–131, Sept 1996.

[13] C. Fourie, W. Perold, and H. Gerber, "Complete monte carlo model description of lumped-element rsfq logic circuits," *Applied Superconductivity, IEEE Transactions on*, vol. 15, no. 2, pp. 384–387, June 2005.

[14] A. Singhee and R. Rutenbar, "Why quasi-monte carlo is better than monte carlo or latin hypercube sampling for statistical circuit analysis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 11, pp. 1763–1776, Nov 2010.

[15] D. P. Kroese, T. Taimre, and Z. I. Botev, *Handbook of Monte Carlo Methods*. Wiley, 2011.

[16] M. Mckay, R. Beckman, and W. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 42, no. 1, Special 40th Anniversary Issue, pp. 56–60, Feb 2000.

[17] I. Dalal, D. Stefan, and J. Harwayne-Gidansky, "Low discrepancy sequences for monte carlo simulations on reconfigurable platforms," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, July 2008, pp. 108–113.

[18] "Boost c++ library," http://www.boost.org/.

[19] I. Sobol, "Uniformly distributed sequences with an additional uniform property," *USSR Computational Mathematics and Mathematical Physics*, vol. 16, no. 5, pp. 236–242, Jan 1975.

[20] P. Bratley and B. L. Fox, "Algorithm 659: Implementing sobol's quasirandom sequence generator," *ACM Trans. Math. Softw.*, vol. 14, no. 1, pp. 88–100, Mar. 1988. [Online]. Available: http://doi.acm.org/10.1145/42288.214372

[21] S. Joe and F. Kuo, "Remark on algorithm 659: Implementing sobol's quasirandom sequence generator," *ACM Trans. Math. Softw.*, vol. 29, no. 1, Mar 2003.

[22] "Sobol sequence generator," http://web.maths.unsw.edu.au/~fkuo/sobol/.

[23] R. Pratap, P. Sen, C. Davis, R. Mukhophdhyay, G. May, and J. Laskar, "Neurogenetic design centering," *Semiconductor Manufacturing, IEEE Transactions on*, vol. 19, no. 2, pp. 173–182, May 2006.

[24] L. Xieting and D. Guojie, "Neural network for yield estimation," in *Circuits and Systems, 1991. Conference Proceedings, China., 1991 International Conference on*, Jun 1991, pp. 298–300 vol.1.

[25] L.-I. Tong, W.-I. Lee, and C.-T. Su, "Using a neural network-based approach to predict the wafer yield in integrated circuit manufacturing," *Components, Packaging, and Manufacturing Technology, Part C, IEEE Transactions on*, vol. 20, no. 4, pp. 288–294, Oct 1997.

[26] G. Creech, J. Zurada, and P. Aronhime, "Feedforward neural networks for estimating ic parametric yield and device characterization," in *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*, vol. 2, Apr 1995, pp. 1520–1523 vol.2.

[27] D. Graupe, *Principles of Artificial Neural Networks*. World Scientific, 2007, vol. 6.

[28] W. McCulloch and W. Pitts, "A logical calculus of the ideas imminent in nervous activity," *Bulletin of Mathematical Biophysics*, no. 6, pp. 115–133, 1943.

[29] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[30] B. Widrow and M. Hoff, "Adaptive switching circuits," *1960 URE WESCON Convention Record*, pp. 96–104, 1960.

[31] F. Rosenblatt, "The perceptron: a probabilistic model of information storage and organization in the brain," *Psychological Review*, no. 65, pp. 368–408, 1958.

[32] P. Picton, *Introduction To Neural Networks*. Macmillan, 1994.

[33] J. C. Principe, E. N.R., and W. Lefebvre, *Neural and Adaptive Systems: Fundamentals Through Simulation*. Wiley, 2000.

[34] "Fast artificial neural networks library," http://leenissen.dk/fann/wp/.

[35] H. Gerber, C. Fourie, W. Perold, and L. Muller, "Design of an asynchronous microprocessor using rsfq-at," *Applied Superconductivity, IEEE Transactions on*, vol. 17, no. 2, pp. 490 –493, june 2007.

[36] P. Bunyk and P. Litskevitch, "Case study in rsfq design: fast pipelined parallel adder," *Applied Superconductivity, IEEE Transactions on*, vol. 9, no. 2, pp. 3714 –3720, jun 1999.

[37] V. Adler, C.-H. Cheah, K. Gaj, D. Brock, and E. Friedman, "A cadence-based design environment for single flux quantum circuits," *Applied Superconductivity, IEEE Transactions on*, vol. 7, no. 2, pp. 3294 –3297, jun 1997.

[38] Y. Kameda, S. Yorozu, and Y. Hashimoto, "Automatic single-flux-quantum (sfq) logic synthesis method for top-down circuit design," *Journal of Physics: Conference Series*, vol. 43, pp. 1179 –1182, 2006.

[39] ——, "A new automatic placement and routing design technology for large-scale single-flux- quantum logic circuits," *Extended Abstract of ISEC 03, Sydney, Australia*, July 2003.

[40] K. Gaj, E. Friedman, and M. Feldman, "Timing of multi-gigahertz rapid single flux quantum digital circuits," *Journal of VLSI Signal Processing*, vol. 16, pp. 247 –276, 1997.

[41] J.-C. Lin and V. Semenov, "Timing circuits for rsfq digital systems," *Applied Superconductivity, IEEE Transactions on*, vol. 5, no. 3, pp. 3472–3477, Sept 1995.

[42] K. Gaj, C.-H. Cheah, E. Friedman, and M. Feldman, "Functional modeling of rsfq circuits using verilog hdl," *Applied Superconductivity, IEEE Transactions on*, vol. 7, no. 2, pp. 3151 –3154, jun 1997.

[43] S. Intiso, I. Kataeva, E. Tolkacheva, H. Engseth, K. Platov, and A. Kidiyarova-Shevchenko, "Time-delay optimization of rsfq cells," *Applied Superconductivity, IEEE Transactions on*, vol. 15, no. 2, pp. 328 – 331, june 2005.

[44] F. Matsuzaki, N. Yoshikawa, M. Tanaka, A. Fujimaki, and Y. Takai, "A behavioral-level hdl description of sfq logic circuits for quantitative performance analysis of large-scale sfq digital systems," *Physica C: Superconductivity*, vol. 392âĂŞ396, Part 2, no. 0, pp. 1495 – 1500, 2003.

[45] H. Toepfer, T. Harnisch, J. Kunert, S. Lange, and H. Uhlmann, "Formal description of the functional behavior of rsfq logic circuits for design and optimization purposes," *Applied Superconductivity, IEEE Transactions on*, vol. 7, no. 2, pp. 3630 –3633, jun 1997.

[46] M. Dorojevets, C. Ayala, and A. Kasperek, "Data-flow microarchitecture for wide datapath rsfq processors: Design study," *Applied Superconductivity, IEEE Transactions on*, vol. 21, no. 3, pp. 787–791, June 2011.

[47] L. Muller and C. Fourie, "Automated state machine and timing characteristic extraction for rsfq circuits," *Applied Superconductivity, IEEE Transactions on*, vol. 24, no. 1, pp. 3–12, Feb 2014.

[48] J. E. Fang and T. Van Duzer, "A josephson integrated circuit simulator (jsim) for superconductive electronic applications," *Extended Abstracts of 4thnd International Superconductive Electronics Conference(ISEC)*, pp. 407–410, 1989.

[49] "Institute of Photonic Technology (IPHT), Jena, Germany," http://www.ipht-jena.de/.

[50] X. Yang, *Engineering Optimization*. John Wiley and Sons, 2010.

[51] Q. Kerr and M. Feldman, "Multiparameter optimization of rsfq circuits using the method of inscribed hyperspheres," *Applied Superconductivity, IEEE Transactions on*, vol. 5, no. 2, pp. 3337–3340, June 1995.

[52] T. Harnisch, J. Kunert, H. Toepfer, and H. Uhlmann, "Design centering methods for yield optimization of cryoelectronic circuits," *Applied Superconductivity, IEEE Transactions on*, vol. 7, no. 2, pp. 3434–3437, June 1997.

[53] C. Fourie and W. Perold, "Comparison of genetic algorithms to other optimization techniques for raising circuit yield in superconducting digital circuits," *Applied Superconductivity, IEEE Transactions on*, vol. 13, no. 2, pp. 511 – 514, june 2003.

[54] Y. Tukel, A. Bozbey, and C. Tunc, "Development of an optimization tool for rsfq digital cell library using particle swarm," *Applied Superconductivity, IEEE Transactions on*, vol. 23, no. 3, pp. 1 700 805–1 700 805, June 2013.

[55] M. Resende and C. Ribeiro, *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Kluwer Academic, 2003.

[56] D. Henderson, S. Jacobson, and A. Johnson, *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Kluwer Academic, 2003.

[57] C. Reeves, *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Kluwer Academic, 2003.

[58] J. Potvin and K. Smit, *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Kluwer Academic, 2003.

[59] M. Celik and A. Bozbey, "A statistical approach to delay, jitter and timing of signals of rsfq wiring cells and clocked gates," *Applied Superconductivity, IEEE Transactions on*, vol. 23, no. 3, pp. 1 701 305–1 701 305, June 2013.

[60] "Inductex website," http://www.inductex.info.